1 സ്ട്രക്ചറും പോയിന്ററും

പ്രധാന പഠനനേട്ടങ്ങൾ

പഠിതാവ് ആർജിക്കേണ്ട ശേഷികൾ

- ഉപയോക്താവിനാൽ നിർവചിക്കപ്പെടുന്ന (user defined) ഡാറ്റാ ഇനത്തിന്റെ ആവശ്യകതയും സ്ട്രക്ചർ ഉപയോഗിച്ച് വിവിധ ഇനം ഡാറ്റ ഏകോപിപ്പിക്കുന്നതിന്റെ സാധ്യതയും തിരിച്ചറിയുക.
- സ്ട്രക്ചർ ഡാറ്റാതരം നിർമിച്ച് അതിലെ അംഗങ്ങളെ റഫർ ചെയ്യുക.
- അടിസ്ഥാന ഡാറ്റകളെയും ഗ്രൂപ്പ് ഡാറ്റകളെയും പ്രതിനിധീകരിക്കുന്നതിൽ നെസ്റ്റഡ് സ്ട്രക്ചറിന്റെ ഉപയോഗം മനസ്സിലാക്കുക.
- ദൈനംദിന ജീവിതത്തിലെ പ്രശ്നങ്ങൾ പരിഹരിക്കാൻ സ്ട്രക്ചർ ഉപയോഗിച്ചുള്ള C++ പ്രോഗ്രാമുകൾ തയാറാക്കുക.
- പോയിന്റർ എന്ന ആശയവും, &, * എന്നീ ഓപ്പറേറ്ററുകളുടെ ഉപയോഗവും വിവരിക്കുക.
- രണ്ട് തരം മെമ്മറി നീക്കിവയ്ക്കലുകൾ താരതമ്യം ചെയ്യുക. ഡൈനാമിക് ഓപ്പറേറ്ററുകളായ `new`, `delete` എന്നീ ഓപ്പറേറ്ററുകളുടെ ഉപയോഗം മനസ്സിലാക്കുക.
- പോയിന്ററിലെ ഓപ്പറേഷനുകൾ ഉദാഹരണം സഹിതം വിവരിക്കുകയും . അതിന്റെ ഔട്ട്പുട്ട് നിർവചിക്കുകയും ചെയ്യുക.
- പോയിന്ററും അറെയും തമ്മിലുള്ള ബന്ധം സമർഥിക്കുക.
- സ്ട്രിങ്ങുകൾ കൈകാര്യം ചെയ്യുന്നതിൽ പോയിന്ററിന്റെ ഉപയോഗം അറിയുക.
- സ്വയം സൂചിത (self referential) സ്ട്രക്ചറിന്റെ ആശയം വിവരിക്കുക.

വിവിധ പ്രശ്നങ്ങൾ നിർധാരണം ചെയ്യാനുള്ള C++ പ്രോഗ്രാമുകൾ പരിചയപ്പെട്ടിട്ടുണ്ടല്ലോ. മിക്കവാറും എല്ലാ പ്രശ്നങ്ങളിലും ഡാറ്റ പ്രോസസ്സിംങ് ആവശ്യമായി വരാറുണ്ട്. അടിസ്ഥാന ഡാറ്റ വിഭാഗത്തിലുള്ള പൂർണസംഖ്യകൾ (integers), ദശാംശ സംഖ്യകൾ (fractional numbers) കാരക്ടറുകൾ, സ്ട്രിംഗുകൾ എന്നിവയെക്കുറിച്ച് നാം കഴിഞ്ഞ വർഷം മനസ്സിലാക്കി. ഈ ഡാറ്റകളെ സൂചിപ്പിക്കുന്നതിന് വേരിയബിളുകൾ ഉപയോഗിക്കുകയും ഇത്തരം വേരിയബിളുകളെ C++ ലെ അടിസ്ഥാന ഡാറ്റഇനങ്ങൾ ഉപയോഗിചച്ച് സ്ഥാപിക്കുകയും ചെയ്തു. എല്ലാ ഡാറ്റയും അടിസ്ഥാന വിഭാഗത്തിൽപെടുന്നില്ല എന്ന് നമുക്കറിയാം. ചിലത് വിവിധതരം അടിസ്ഥാന ഡാറ്റ വിഭാഗങ്ങളുടെ ഏകോപനത്തിലൂടെയാണ് നിർമിക്കപ്പെടുന്നത്. എല്ലാത്തരം ഡാറ്റകളെയും ഉൾക്കൊള്ളുന്ന തരത്തിൽ ഡാറ്റാ ടൈപ്പുകൾ നിർണയിക്കാൻ ഒരു പ്രോഗ്രാമിങ് ഭാഷയ്ക്കും സാധിക്കില്ല. അതുകൊണ്ടുതന്നെ പ്രോഗ്രാമിങ് ഭാഷകളിൽ ഉപയോക്താവിന്റെ ആവശ്യമനുസരിച്ച് പുതിയ ഡാറ്റാഇനങ്ങൾ നിർമിക്കാനുള്ള സൗകര്യം അനുവദിച്ചിട്ടുണ്ട്. ഈ അധ്യായത്തിൽ നമ്മൾ അത്തരത്തിലുള്ള ഉപയോക്ത്യ നിർവചിത ഡാറ്റാഇനം (user-defined data type) ആയ സ്ട്രക്ചർ (structure) എന്ന ഡാറ്റാ ടൈപ്പിനെ കുറിച്ച് ചർച്ച ചെയ്യുന്നു. മറ്റൊരു തരം വേരിയബിളായ പോയിന്ററിനെ (pointer) കുറിച്ചും ഈ അധ്യായത്തിൽ പരിചയപ്പെടാം. പോയിന്റർ എന്ന ആശയം C, C++ എന്നീ ഭാഷകളിലെ ഒരു പ്രത്യേകതയാണ്. ഇത് മെമ്മറി സ്ഥാനങ്ങളെ നേരിട്ട്

അതിന്റെ വിലാസത്തിലൂടെ (Address) പ്രാപ്യമാക്കുകയും അതുവഴി പ്രോഗ്രാം നടപ്പിലാക്കൽ (Execution) വേഗത്തിലാക്കുകയും ചെയ്യുന്നു. ഈ ആശയം നന്നായി മനസ്സിലാക്കിയാൽ നമുക്ക് സിസ്റ്റം ലെവൽ (system level) പ്രോഗ്രാമുകളും ഡാറ്റാ സ്ട്രക്ചർ ആപ്ലിക്കേഷനുകളും എളുപ്പത്തിൽ തയാറാക്കാം.

നമുക്ക് ചെയ്യാം

കമ്പ്യൂട്ടർ ഹാർഡ്വെയറുകളെ നിയന്ത്രിക്കുന്ന പ്രോഗ്രാമുകളെ സിസ്റ്റം ലെവൽ പ്രോഗ്രാം എന്നു വിളിക്കാം. സ്റ്റാക്ക്, ക്യൂ, ലിങ്ക്ഡ് ലിസ്റ്റ് തുടങ്ങിയ ഡാറ്റാ രൂപങ്ങളെ നിർമിക്കാനും ഉപയോഗിക്കാനുമുള്ള പ്രോഗ്രാമുകളെ ഡാറ്റാ സ്ട്രക്ച്ചർ ആപ്ലിക്കേഷൻ എന്ന് വിളിക്കാം.

നമ്മൾ കഴിഞ്ഞ വർഷം ഗ്നു കമ്പയിലർ കളക്ഷൻ (GCC) ഉൾക്കൊള്ളുന്ന ജിനി IDE ആണ് C++ പ്രോഗ്രാം നിർമിക്കാൻ ഉപയോഗിച്ചത്. ഈ അധ്യായത്തിൽ C++ പ്രോഗ്രാമുകൾ പൂർണമായും GCC ഉപയോഗിച്ച് ചെയ്യാവുന്ന തരത്തിലാണ് അവതരിപ്പിച്ചിട്ടുള്ളത്.

1.1 സ്ട്രക്ചറുകൾ

വിദ്യാർഥികൾ, തൊഴിലാളികൾ, ഉദ്യോഗസ്ഥർ തുടങ്ങിയവർ അവരവരുടെ സ്ഥാപനങ്ങൾ നൽകുന്ന തിരിച്ചറിയൽ കാർഡ് (Identity card) ധരിക്കാറുണ്ട്. ചിത്രം 1.1 കാണിക്കുന്നത് ഒരു കുട്ടിയുടെ തിരിച്ചറിയൽ കാർഡ് ആണ്. പട്ടിക 1.1 ലെ ഒന്നാമത്തെ കോളത്തിൽ കാർഡിൽ പ്രിന്റ് ചെയ്തിരിക്കുന്ന ചില ഡാറ്റകളാണ് കാണിച്ചിരിക്കുന്നത്. പട്ടികയുടെ രണ്ടാമത്തെ കോളം ഈ ഡാറ്റകൾക്കനുയോജ്യമായ C++ ഡാറ്റാഇനങ്ങൾ ഉപയോഗിച്ച് പൂരിപ്പിക്കുക.

ചിത്രം1.1: കുട്ടിയുടെ തിരിച്ചറിയൽ കാർഡ്

Data	C++ data type
12345	
Sneha S. Raj	
20/02/1997	
O+ve	
Snehanilayam, Gandhi Nagar, Chemmanavattom, Pin 685 531	

പട്ടിക 1.1: ഡാറ്റയും C++ ഡാറ്റാ ടൈപ്പുകളും

നിങ്ങൾ അഡ്മിഷൻ നമ്പറിന് `short` അല്ലെങ്കിൽ `int` ഡാറ്റാടൈപ്പും, പേര് (Sneha S. Raj), രക്തഗ്രൂപ്പ് (blood group (O +ve)), വിലാസം (അഡ്രസ്സ്) തുടങ്ങിയവയ്ക്ക് `char` അറെയും ഉപയോഗിച്ചിട്ടുണ്ടാവും. ചിലപ്പോൾ നിങ്ങൾക്ക് ജനന തീയതിക്കും (Date of birth) വിലാസത്തിനും ഏറ്റവും യോജിച്ച ഡാറ്റാഇനം കണ്ടെത്താൻ കഴിഞ്ഞില്ല എന്നു വന്നേക്കാം. നമുക്ക് 20/02/1997 എന്ന തീയതി പരിഗണിക്കാം. ഇത് ഒരു സംയോജിത (compound) ഡാറ്റാ ഇനമാണ് എന്ന് വിശകലനം ചെയ്താൽ മനസ്സിലാവും. ഇതിൽ

ദിവസസംഖ്യ (20), മാസസംഖ്യ (02), വർഷസംഖ്യ (1997) എന്നിവ കൂടിച്ചേർന്നിരിക്കുന്നു. മാസസംഖ്യക്ക് പകരം മാസത്തിന്റെ പേര് ഉപയോഗിക്കാറുണ്ട്. ഇതുപോലെ വിലാസം എന്നത്, വീട്ടുനമ്പർ, വീട്ടുപേര്, സ്ഥലം, ജില്ല, സംസ്ഥാനം, പിൻകോഡ് എന്നിവയുടെ ഒരു സംയോജിത രൂപമാണ്. ഈ കാർഡിലെ മുഴുവൻ വിവരങ്ങളെയും വേണമെങ്കിൽ ഒരു വലിയ സംയോജിത ഡാറ്റാഇനമായി (grouped datatype) പരിഗണിക്കാം. ഇത്തരം സംയോജിത ഡാറ്റാതരത്തിനെ പ്രതിനിധാനം ചെയ്യാൻ ഉപയോക്താവ് നിർവചിക്കുന്ന ഡാറ്റാടൈപ്പിനെ ഉപയോക്തൃ നിർവചിത ഡാറ്റാഇനം (user defined data type) എന്നു വിളിക്കുന്നു. C++ ൽ ഇത്തരം പുതിയ ഡാറ്റ ടൈപ്പുകൾ നിർവചിക്കാനുള്ള സൗകര്യമുണ്ട്.

സ്ട്രക്ചർ: - യുക്തിപരമായി പരസ്പരം ബന്ധപ്പെട്ടു നിൽക്കുന്ന ഒരു കൂട്ടം ഡാറ്റാ ഇനങ്ങളെ പ്രതിനിധാനം ചെയ്യാൻ C++ ൽ ഉപയോഗിക്കുന്ന പൊതു പേരോടുകൂടിയ ഉപയോക്തൃ നിർവചിത ഡാറ്റ ഇനമാണ് സ്ട്രക്ചർ. ഈ ഡാറ്റകൾ ചിലപ്പോൾ വ്യത്യസ്തങ്ങളായ ഡാറ്റാഇനങ്ങളിൽ ഉള്ളവ ആയിരിക്കും. 11-ാം ക്ലാസിൽ നമ്മൾ പഠിച്ച അറെ ഒരേ തരത്തിലുള്ള ഡാറ്റയുടെ ഏകോപിതരൂപമാണ്. പക്ഷേ ഒരു പേരിലൂടെ അറിയപ്പെടുന്ന പല തരത്തിലുള്ള ഡാറ്റയുടെ ഏകോപിത രൂപമാണ് സ്ട്രക്ചർ. C++ ൽ സ്ട്രക്ചർ നിർവചിക്കുന്നതും ഉപയോഗിക്കുന്നതും നമുക്ക് ചർച്ച ചെയ്യാം.

1.1.1 സ്ട്രക്ചർ നിർവചനം

കമ്പ്യൂട്ടർ പ്രോഗ്രാമിലൂടെ പ്രശ്നങ്ങൾ നിർധാരണം ചെയ്യുമ്പോൾ മേൽ വിവരിച്ച പ്രകാരം ഏകോപിത ഡാറ്റാഇനങ്ങൾ ഉപയോഗിക്കേണ്ടിവരും. ആ സമയത്ത് ഏകോപിത ഡാറ്റാഇനത്തിന് കൃത്യമായ പേര് നൽകുന്നതോടൊപ്പം, അതിൽ അടങ്ങിയിരിക്കുന്ന ഡാറ്റാഇനങ്ങൾ കണ്ടെത്തി നിർവചിക്കേണ്ടതുണ്ട്. താഴെ കൊടുത്തിരിക്കുന്ന വാക്യഘടന ഉപയോഗിച്ച് സ്ട്രക്ചറിനെ നിർവചിക്കാം:

```
struct structure_tag
{
    data_type variable1;
    data_type variable2;
    ..................;
    ..................;
    data_type variableN;
};
```

മുകളിൽ ചേർത്ത വാക്യഘടനയിൽ **struct** എന്നത് സ്ട്രക്ചർ നിർവചിക്കാനുള്ള കീവേഡ് ആണ്. structure_tag സ്ട്രക്ചറിന്റെ പേര് നിർണയിക്കുന്ന ഐഡന്റിഫയർ ആണ്. variable1, variable2, ..., variableN എന്നിവ സ്ട്രക്ചറിലെ അടിസ്ഥാന ഡാറ്റാ ഘടകങ്ങളുടെ പേര് നിർണയിക്കുന്ന ഐഡന്റിഫയറുകളാണ്. സ്ട്രക്ചറിന്റെ പേര് സൂചിപ്പിക്കാൻ ഉപയോഗിക്കുന്ന ഐഡന്റിഫയർ ആയ structure_tag ഒരു പുതിയ ഉപയോക്തൃ നിർവചിത ഡാറ്റാഇനമാണ്. ഈ ഡാറ്റാഇനത്തെയും മറ്റേ

തൊരു ഡാറ്റ ഇനത്തെയും പോലെ നിശ്ചിത മെമ്മറി ആവശ്യമാണ്. ഇത് പുതിയ വേരിയബിളുകൾ പ്രഖ്യാപിക്കാൻ ചെയ്യാൻ ഉപയോഗിക്കുന്നു. ഈ ഡാറ്റാഇനത്തിലുള്ള വേരിയബിളുകൾ ഫങ്ഷൻ ആർഗ്വിമെന്റായോ, ഫങ്ഷനിൽ നിന്നും റിട്ടേൺ ചെയ്യുന്ന വേരിയബിളായോ ഉപയോഗിക്കാം. സ്ട്രക്ചർ നിർവചനത്തിൽ ബ്രായ്ക്കറ്റിനുള്ളിൽ ({ }) കാണുന്ന വേരിയബിളുകളെ സ്ട്രക്ചറിന്റെ മെമ്പർ വേരിയബിളുകൾ എന്നു പറയുന്നു. ഇത്തരം മെമ്പർ വേരിയബിളിന്റെ ഡാറ്റാഇനം അടിസ്ഥാന ഡാറ്റാഇനമോ ഉപയോക്ത്യ നിർവചിത ഡാറ്റാഇനമോ ആവാം. മെമ്പർ വേരിയബിളുകളുടെ വലുപ്പങ്ങളുടെ തുകയാണ് സ്ട്രക്ചർ വേരിയബിളിന്റെ ആകെ വലുപ്പം.

ഇനി നമുക്ക് ഐഡന്റിറ്റി കാർഡിലെ ജനനത്തീയതിയായ 20/02/1997 പ്രതിനിധീകരിക്കുന്ന ഒരു സ്ട്രകചർ തയാറാക്കി നോക്കാം. ഈ തീയതിയിലൽ മൂന്ന് പൂർണ സംഖ്യകൾ ഉണ്ട്. ആദ്യത്തേത് ദിവസസംഖ്യയും രണ്ടാമത്തേത് മാസസംഖ്യയും മൂന്നാമത്തേത് വർഷസംഖ്യയുമാണ്. ചുവടെചേർത്ത രീതിയിൽ നമുക്ക് ഒരു സ്ട്രക്ചർ ഘടന തയാറാക്കാം.

```
struct date
{
    int dd;
    int mm;
    int yy;
};
```

ഇവിടെ date എന്നത് സ്ട്രക്ചർ ടാഗിനെ (സ്ട്രക്ച്ചറിന്റെ പേരിനെ) സൂചിപ്പിക്കുന്നു. dd, mm, yy എന്നീ int ടൈപ്പ് വേരിയബിളുകൾ യഥാക്രമം ദിവസം, മാസം, വർഷം എന്നിവയെ സൂചിപ്പിക്കുന്നു. ഇവ സ്ട്രക്ചറിലെ അംഗങ്ങളാണ്. മാസസംഖ്യയ്ക്ക് പകരം മാസത്തിന്റെ പേര് ചേർക്കണമെങ്കിൽ സ്ട്രക്ചർ നിർവചനത്തിൽ താഴെ കൊടുത്ത രീതിയിൽ മാറ്റം വരുത്തണം.

```
struct strdate
{
    int day;
    char month[10];      //  മാസത്തിന്റെ പേര് സ്ട്രിങ്ങാണ്
    int year;
};
```

പ്രോഗ്രാമുകൾ നിർമിക്കുന്ന സമയത്ത് ചില ഡാറ്റകൾ പരസ്പരബന്ധം പുലർത്തുന്നതായി കാണാം. ഈ ഡാറ്റകളെ ഏകോപിപ്പിച്ച് കുറച്ചുകൂടി ഒതുക്കമുള്ള രൂപത്തിലേക്ക് മാറ്റാൻ സ്ട്രക്ചറുകൾ ഉപയോഗിക്കാം. ഉദാഹരണത്തിന് അഡ്മിഷൻ നമ്പർ, പേര്, ഗ്രൂപ്പ്, ഫീസ് തുടങ്ങിയ വിവരങ്ങൾ കുട്ടിയുമായി ബന്ധപ്പെട്ടിരിക്കുന്നു. താഴെ കൊടുത്ത രീതിയിൽ ഒരു സ്ട്രക്ചർ നിർമിക്കാം.

```
struct student
{
    int adm_no;
```

```
    char name[20];
    char group[10];
    float fee;
};
```

നമുക്ക് ചെയ്യാം

ഇനി നിങ്ങൾ സ്വന്തമായി സ്ട്രക്ചർ നിർവചിക്കാൻ ശ്രമിക്കൂ. **'address'**ഉം **'blood group'** ഉം ആകട്ടെ പുതിയ സ്ട്രക്ചറുകൾ. ബ്ലഡ് ഗ്രൂപ്പിൽ, ഗ്രൂപ്പിന്റെ പേരും, **rh** വിലയും ഉണ്ടായിരിക്കണം.

തൊഴിലാളിയുടെ വിവരങ്ങളിൽ **employee code, name, gender, designation, salary** എന്നിവ പ്രധാനപ്പെട്ടവയാണ് എന്ന് നമുക്കറിയാമല്ലോ. ഇവ ഉൾപ്പെടുത്തി അനുയോജ്യമായ ഒരു സ്ട്രക്ചർ നിർവചിക്കുക.

ഇതുവരെ നമ്മൾ ചർച്ച ചെയ്തത് സ്ട്രക്ചർ നിർവചിക്കുന്ന രീതിയാണ്. ഇനി സ്ട്രക്ചർ വേരിയബിളുകൾ എങ്ങനെ പ്രഖ്യാപിക്കാമെന്നും അതിൽ ഡാറ്റ എങ്ങനെ ശേഖരിക്കാമെന്നും മനസ്സിലാക്കാം.

1.1.2 വേരിയബിൾ പ്രഖ്യാപനവും (Declaration) മെമ്മറി നീക്കിവയ്ക്കലും (Allocation)

അടിസ്ഥാന ഡാറ്റാഇനങ്ങളെ പോലെ തന്നെ സ്ട്രക്ചർ ഡാറ്റാഇനത്തിനും വേരിയബിളുകൾ ഉണ്ടെങ്കിൽ മാത്രമേ ഡാറ്റ ശേഖരിക്കാൻ കഴിയൂ. താഴെ കൊടുത്തിരിക്കുന്ന മാതൃകയിലാണ് വേരിയബിളുകൾ പ്രഖ്യാപിക്കുന്നത്.

```
struct structure_tag var1, var2, ..., varN;
```

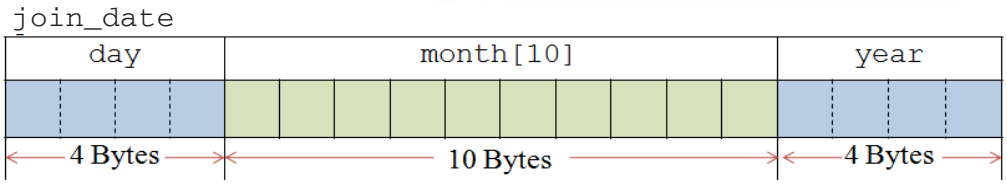
OR

```
structure_tag var1, var2, ..., varN;
```

ഈ വാക്യഘടനയിൽ `structure_tag` എന്നത് സ്ട്രക്ചറിന്റെ പേരും `var1, var2, ..., varN` എന്നിവ സ്ട്രക്ചർ വേരിയബിളുകളുമാണ്. നമുക്ക് `date, strdate` എന്നീ സ്ട്രക്ചറുകളിൽ വേരിയബിളുകൾ പ്രഖ്യാപിക്കാം.

```
date dob, today;          OR   struct date dob, today;
strdate adm_date, join_date;
```

വേരിയബിൾ പ്രഖ്യാപനം നടക്കുമ്പോൾ അതിന്റെ വലുപ്പം അനുസരിച്ച് മെമ്മറി നീക്കിവയ്ക്കുന്നു എന്ന് നമുക്കറിയാം. ഒരു സ്ട്രക്ചർ വേരിയബിളിന്റെ വലുപ്പം എന്തായിരിക്കും? സ്ട്രക്ചർ ഒരു ഉപയോക്തൃ നിർവചിത ഡാറ്റാഇനമായതിനാൽ അതിന്റെ വലുപ്പം, നിർവചനം അനുസരിച്ചായിരിക്കും. മേൽ സൂചിപ്പിച്ച വേരിയബിൾ പ്രഖ്യാപനത്തിൽ, സ്ട്രക്ച്ചർ `date` വേരിയബിളുകളായ `dob` യും `today` യും 12 byte വീതം വലുപ്പമുള്ളവയായിരിക്കും (മൂന്ന് `int` ഡാറ്റാഇനത്തിലുള്ള അംഗങ്ങൾ, GCC യിൽ `int` ന്റെ വലുപ്പം 4 byte ആയതിനാൽ 3 x 4 = 12 bytes). സ്ട്രക്ചർ `strdate` ലെ വേരിയബിളായ `join_date` ന്റെ മെമ്മറി ഘടന ചിത്രത്തിൽ ചേർക്കുന്നു.



ചിത്രം 1.2: സ്ട്രക്ചർ വേരിയബിളിന്റെ മെമ്മറി നീക്കിവയ്ക്കൽ

GCC യിൽ int ടൈപ്പിന്റെ വലുപ്പം 4 bytes ഉം Turbo IDE യിൽ 2 bytes ഉം ആണ്. ചിത്രം 1.2 ൽ day, year എന്നീ അംഗങ്ങൾ 4 bytes വീതം ഉപയോഗിക്കുന്നു. GCC ഉപയോഗിക്കുന്നതിനാൽ ചിലപ്പോൾ നമ്മൾക്ക് ഇത്രയും മെമ്മറി ആവശ്യമില്ല എങ്കിൽ short ഡാറ്റാഇനം ഉപയോഗിക്കുകയാണ് നല്ലത്.

join_date എന്ന വേരിയബിളിൽ day, month, year എന്നീ മൂന്ന് അംഗങ്ങളാണ് ഉള്ളത്. ഇവ യഥാക്രമം 4 bytes, 10 bytes, 4 bytes മെമ്മറി ഉപയോഗിക്കുന്നു. അങ്ങനെ ഈ വേരിയബിളിന്റെ ആകെ മെമ്മറി ഉപയോഗം 18 bytes ആണ്.

നമുക്ക് ചെയ്യാം

ഇനി നമ്മൾ മുമ്പ് നിർവചിച്ച സ്ട്രക്ചർ **student** ന്റെ മെമ്മറി വലുപ്പം കണ്ടുപിടിക്കുക. കൂടാതെ **student** സ്ട്രക്ചറിൽ വേരിയബിൾ പ്രഖ്യാപിക്കുക. ഈ വേരിയബിളിന്റെ മെമ്മറി നീക്കി വയ്ക്കുന്നതിന്റെ ചിത്രം വരയ്ക്കുക.

താഴെ കൊടുത്തിരിക്കുന്ന രീതിയിൽ ഒരു സ്ട്രക്ചർ വേരിയബിളിനെ അതിന്റെ നിർവചനത്തിന്റെ കൂടെത്തന്നെ പ്രഖ്യാപിക്കാം.

```
struct complex
{
    short real;
    short imaginary;
}c1, c2;
```

complex എന്നു പേരുള്ള സ്ട്രക്ചർ കോംപ്ലക്സ് നമ്പറിനെ പ്രതിനിധാനം ചെയ്യുന്നു. c1 ഉം c2 ഉം ഈ സ്ട്രക്ചർ തരത്തിലുള്ള രണ്ട് വേരിയബിളുകളാണ്. സ്ട്രക്ചർ നിർവചനത്തിന്റെ കൂടെ വേരിയബിൾ പ്രഖ്യാപിക്കുമ്പോൾ വേണമെങ്കിൽ സ്ട്രക്ചർ ടാഗ് ഒഴിവാക്കുകയും ചെയ്യാം. ചുവടെ കൊടുത്ത ഉദാഹരണം നോക്കുക.

```
struct
{
    int a, b, c;
}eqn_1, eqn_2;
```

ഈ നിർവചനത്തിനു ഒരു പരിമിതിയുണ്ട്. സ്ട്രക്ചർ ടാഗ് ഇല്ലാത്തതിനാൽ വീണ്ടും പുതിയ വേരിയബിളുകൾ നിർമിക്കാൻ ഈ നിർവചനം പര്യാപ്തമല്ല. സ്ട്രക്ചറിലെ അംഗങ്ങൾ ഒരേ തരമാണെങ്കിൽ ഒറ്റവരി പ്രഖ്യാപനമേ ആവശ്യമുള്ളൂ.

വേരിയബിളിന് പ്രാരംഭ വിലനൽകൽ (Initialisation)

വേരിയബിളുകൾ പ്രഖ്യാപിക്കുമ്പോൾത്തന്നെ അതിലേക്ക് ചില വിലകൾ നൽകാവുന്നതാണ്. ഇത് സ്ട്രകചർ വേരിയബിളിന്റെ കാര്യത്തിലും ശരിയാണ്. സ്ട്രക്ചർ വേരിയബിളിന് താഴെ കൊടുത്ത രീതിയിൽ പ്രാരംഭവില നൽകാം.

```
structure_tag variable={value1, value2,..., valueN};
```

ഉദാഹരണമായി student സ്ട്രക്ചറിന് താഴെ കൊടുത്ത രീതിയിൽ പ്രാരംഭവില നൽകാം.

```
student s={3452, "Vaishakh", "Science", 270.00};
```

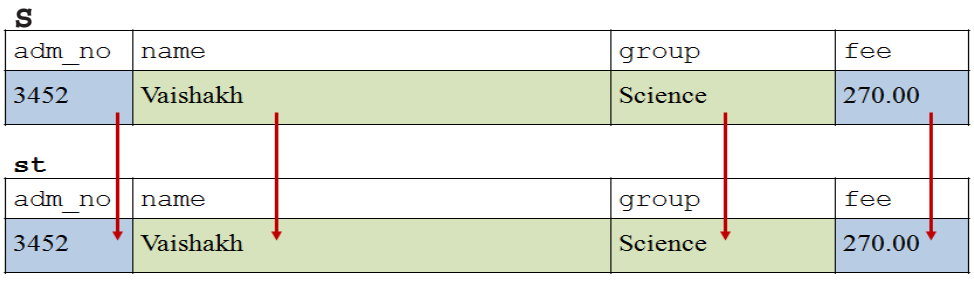
സ്ട്രക്ചർ അംഗങ്ങളായ വേരിയബിളുകൾക്ക് നിർവചനത്തിൽ നൽകിയിരിക്കുന്ന അതേ ക്രമത്തിലാണ് ബ്രായ്ക്കറ്റിൽ കൊടുത്ത ഡാറ്റകൾ നൽകിയിരിക്കുന്നത്. അതുകൊണ്ടു തന്നെ ഡാറ്റയുടെ ക്രമത്തിലും തരത്തിലും ശ്രദ്ധിക്കേണ്ടതാണ്. മുകളിൽ കൊടുത്ത C++ വാചകം 38 ബൈറ്റ്സ് മെമ്മറി s എന്ന വേരിയബിളിനു വേണ്ടി ഉപയോഗിക്കുന്നു. 3452, "Vaishakh", "Science", 270.00 എന്നീ ഡാറ്റ adm_no, name, group, fee എന്നീ വേരിയബിളുകൾക്ക് യഥാക്രമം നൽകപ്പെടുന്നു (Assign ചെയ്യപ്പെടുന്നു).

എല്ലാ അംഗങ്ങൾക്കും വിലകൾ നൽകിയില്ലെങ്കിൽ ആദ്യം മുതലുള്ള അംഗങ്ങൾക്ക് ക്രമത്തിൽ വിലകൾ നൽകപ്പെടും, അവസാനം ബാക്കി വരുന്ന അംഗങ്ങൾക്ക് സംഖ്യയാണെങ്കിൽ 0 (zero) യും, സ്ട്രിംഗ് ആണെങ്കിൽ '\0' (നൾ ക്യാരക്ടറും) നൽകപ്പെടുകയും ചെയ്യുന്നു.

ഒരു സ്ട്രക്ചർ വേരിയബിളിലേക്ക് മറ്റൊരു സ്ട്രക്ചർ വേരിയബിളിന്റെ ഡാറ്റ നൽകാൻ കഴിയും. അങ്ങനെ ചെയ്യണമെങ്കിൽ രണ്ട് വേരിയബിളുകളും ഒരേ സ്ട്രക്ചർ ടൈപ്പിലുള്ളവ ആയിരിക്കണം. ഉദാഹരണം താഴെ ചേർക്കുന്നു.

```
student st = s;
```

ഈ വാചകം വേരിയബിൾ st ലേക്ക് വേരിയബിൾ 's' ന്റെ ഡാറ്റ ശേഖരിക്കുന്നു.



ചിത്രം 1.3: സ്ട്രക്ചർ അസൈൻമെന്റ്

ഒരു സ്ട്രക്ചർ നിർവചിക്കുമ്പോൾ അതിനകത്തുള്ള അംഗങ്ങളായ വേരിയബിളുകൾക്ക് പ്രാരംഭ വില നൽകാൻ പാടില്ല. കാരണം, സ്ട്രക്ചർ നിർവചനം നടക്കുമ്പോൾ മെമ്മറി നീക്കിവയ്ക്കപ്പെടുന്നില്ല. അതായത്, സ്ട്രക്ച്ചർ നിർവചനത്തിനെ ഒരു വീടിന്റെ ബ്ലൂപ്രിന്റ് (പ്ലാൻ) ആയി കണക്കാക്കാം. പ്ലാനിൽ മുറിയുടെ പേരും രൂപവും കാണാമെങ്കിലും മുറികളിൽ ഒന്നും ശേഖരിക്കാൻ കഴിയില്ല. കാരണം, അവയ്ക്ക് യഥാർഥ സ്ഥലം ഇല്ല എന്നതുതന്നെ. ഒരു വീടിന്റെ ആകെ സ്ഥലം എന്നത് എല്ലാ മുറികളുടേയും സ്ഥലത്തിന്റെ ആകെത്തുകയാണ്. അതുപോലെ സ്ട്രക്ചർ വേരിയബിളിന്റെ മെമ്മറി വലുപ്പം എന്നത് അതിന്റെ അംഗങ്ങളുടെ വലുപ്പത്തിന്റെ തുകയാണ്. ഈ പ്ലാൻകൊണ്ട് ഒരേ രീതിയിലുള്ള കുറേ വീടുകൾ നിർമിക്കാം. വീടുകൾക്ക് വ്യത്യസ്തമായ പേരും സ്ഥലവും ഉണ്ടാകാം. ഇതുപോലെ സ്ട്രക്ചർ നിർവചനം എന്നത് ബ്ലൂപ്രിന്റും സ്ട്രക്ചർ വേരിയബിൾ എന്നത് ഇതിന്റെ പൂർത്തീകരണവുമാണ്. ഒരു ബ്ലൂപ്രിന്റ് ഉപയോഗിച്ച് അനേകം വീടുകൾ നിർമിക്കുന്നതുപോലെ സ്ട്രക്ചർ നിർവചനം ഉപയോഗിച്ച് വ്യത്യസ്തമായ പേരിൽ അനേകം വേരിയബിളുകൾ പ്രഖ്യാപിക്കാം. ഓരോന്നിനും വ്യത്യസ്തമായ മെമ്മറി സ്ഥലവും ഉണ്ടായിരിക്കും.

1.1.3 സ്ട്രക്ചർ അംഗങ്ങളെ ഉപയോഗിക്കൽ

അറെ (array) എന്നത് ഡാറ്റയുടെ കൂട്ടമാണ്. ഒരു അറെയിലെ അംഗങ്ങളെ സബ്സ്ക്രിപ്റ്റുപയോഗിച്ച് നമുക്ക് ശേഖരിക്കാനും എടുക്കാനും കഴിയും. ഇതുപോലെ സ്ട്രക്ചർ അംഗങ്ങളെ ഉപയോഗിക്കാൻ വേണ്ടി പിരീഡ് ചിഹ്നം അഥവാ ഡോട്ട് ഓപ്പറേറ്റർ (.) ഉപയോഗിക്കുന്നു. ഇത് ഉപയോഗിക്കുന്നതിന്റെ വാക്യഘടന ചുവടെ കൊടുത്തിരിക്കുന്നു.

```
structure_variable.element_name
```

പ്രോഗ്രാമുകളിൽ സ്ട്രക്ചർ ഡാറ്റ ഉപയോഗിക്കണമെങ്കിൽ മുകളിൽ കൊടുത്തപോലെ സ്ട്രക്ചർ അംഗങ്ങളെ സൂചിപ്പിക്കണം. ചില ഉദാഹരണങ്ങൾ ചുവടെ കൊടുത്തിരിക്കുന്നു.

```
today.dd = 10;
strcpy(adm_date.month, "June");
cin >> s1.adm_no;
cout << c1.real + c2.real;
```

എന്നാൽ c1+c2 എന്ന പ്രയോഗം സാധ്യമല്ല. കാരണം '+' എന്ന ഓപ്പറേറ്റർ സംഖ്യകളുടെ കൂടെ മാത്രമേ ഉപയോഗിക്കാൻ പാടുള്ളൂ.

ഇനി നമുക്ക് രണ്ട് സ്ട്രക്ചർ വേരിയബിളിലെ വിലനൽകൽ ഓപ്പറേഷനിലെ (Assignment) രസകരമായ ചില കാര്യങ്ങൾ മനസ്സിലാക്കാം.

ചുവടെ ചേർത്തിരിക്കുന്ന രണ്ട് സ്ട്രക്ചറുകൾ പരിഗണിക്കുക.

```
struct test_1
{
    int a;
    float b;
}t1={3, 2.5};
```

```
struct test_2
{
    int a;
    float b;
}t2;
```

മുകളിൽ കൊടുത്ത രണ്ട് സ്ട്രക്ചറുകളിലെയും അംഗങ്ങൾ ഒരേ എണ്ണവും ഒരേ തരവും, ഒരേ പേരുള്ളവയുമാണ്. test_1 ലെ വേരിയബിളായ t1 ലെ അംഗങ്ങളായ a യ്ക്കും b യ്ക്കും 3, 2.5 എന്നീ പ്രാരംഭ വിലകൾ യഥാക്രമം നൽകിയിട്ടുണ്ട്. പക്ഷെ t2=t1; എന്ന വിലനൽകൽ നടത്തുകയാണെങ്കിൽ അത് തെറ്റാണ് എന്ന സന്ദേശം നമുക്ക് ലഭിക്കും. കാരണം t1 ഉം t2 ഉം രണ്ട് തരത്തിലുള്ള സ്ട്രക്ചർ വേരിയബിളുകളാണ്. പക്ഷെ നമ്മൾക്ക് t1 സ്ട്രക്ചറിലെ വില t2 സ്ട്രക്ചറിൽ നൽകണമെങ്കിൽ ചുവടെ ചേർത്ത രീതി ഉപയോഗിക്കാം.

```
t2.a = t1.a;        t2.b = t1.b;
```

ഇത് സാധ്യമാവാൻ കാരണം, int ടൈപ്പിലുള്ള വേരിയബിളുകളാണ് ഇവിടെ വിലനൽകൽ പ്രസ്താവനയിൽ (Assignment statement) ഉപയോഗിച്ചിരിക്കുന്നത്.

ഇനി നമുക്ക് ഇതുവരെ പഠിച്ച കാര്യങ്ങൾ ഉപയോഗിച്ചുള്ള ഒരു പ്രോഗ്രാം പരിചയപ്പെടാം. ഇവിടെ student സ്ട്രക്ചറിൽ രജിസ്റ്റർ നമ്പർ, പേര്, കുട്ടിയുടെ CE മാർക്ക് (തുടർ മൂല്യനിർണയം), PE മാർക്ക് (പ്രാക്ടിക്കൽ മൂല്യനിർണയം), TE (ടേം മൂല്യനിർണയം) എന്നിവ അംഗങ്ങളായി ചേർത്തിട്ടുണ്ട്. സ്ട്രക്ചറിലെ ഈ അംഗങ്ങളുടെ വിലകൾ ഇൻപുട്ടായി നൽകിയാൽ പ്രോഗ്രാം തുടർച്ചയും സമഗ്രവുമായ മൂല്യനിർണയത്തിന്റെ ഭാഗമായുള്ള ആകെ മാർക്ക് കണ്ടുപിടിച്ച് എല്ലാ വിവരങ്ങളും പ്രദർശിപ്പിക്കും.

Program 1.1: കുട്ടിയുടെ ആകെ മാർക്ക് കാണാൻ

```
#include <iostream>
#include <cstdio>      //gets() ഫങ്ഷൻ ഉപയോഗിക്കാൻ
using namespace std;
struct student  //സ്ട്രക്ചർ നിർവചനം തുടങ്ങുന്നു
{
    int reg_no; //  രജിസ്റ്റർ നമ്പർ 32767 ലും കൂടാം, അതുകൊണ്ട് int ഉപയോഗിക്കുന്നു.
    char name[20];
    short ce;//int 4 ബൈറ്റ് ഉപയോഗിക്കും എന്നാൽ ce സ്കോറിന് ചെറിയ സംഖ്യ മതി
    short pe;
    short te;
};                      //സ്ട്രക്ചർ നിർവചനത്തിന്റെ അവസാനം
int main()
{
    student s;          //സ്ട്രക്ചർ വേരിയബിളിന്റെ പ്രഖ്യാപനം
    int tot_score;
    cout<<"Enter register number: ";
    cin>>s.reg_no;
    fflush(stdin);      //കീബോർഡ് ബഫർ ശൂന്യമാക്കാൻ
    cout<<"Enter name: ";
    gets(s.name);
```

```
    cout<<"Enter scores in CE, PE and TE: ";
    cin>>s.ce>>s.pe>>s.te;
    tot_score=s.ce+s.pe+s.te;
    cout<<"\nRegister Number:  "<<s.reg_no;
    cout<<"\nName of Student:  "<<s.name;
    cout<<"\nCE Score:  "<<s.ce<<"\tPE Score:  "<<s.pe
     <<"\tTE Score:  "<<s.te;
    cout<<"\nTotal Score      :  "<<tot_score;
    return 0;
}
```

പ്രോഗ്രാം 1.1 ന്റെ ഔട്ട്പുട്ട് താഴെ ചേർക്കുന്നു.:

ഔട്ട്പുട്ട്:

```
Enter register number: 23545
Enter name: Deepika Vijay
Enter scores in CE, PE and TE: 19   38   54

Register Number:  23545
Name of Student:  Deepika Vijay
CE Score:  19 PE Score:  38     TE Score:  54
Total Score      :  111
```

പ്രോഗ്രാം 1.1 ൽ സ്ട്രക്ചർ നിർവചിച്ചിരിക്കുന്നത് `main()` ഫങ്ഷന് പുറത്താണ്. ഇതുവേണമെങ്കിൽ `main()` ഫങ്ഷന് അകത്തും ചെയ്യാം. നിർവചനത്തിന്റെ സ്ഥാനം സ്ട്രക്ച്ചറിന്റെ പരിധിയും (scope) ആയുസ്സും (life) നിർണയിക്കുന്നു. നമ്മൾ 11-ാം ക്ലാസിലെ 10-ാം അധ്യായത്തിൽ പഠിച്ച ലോക്കൽ, ഗ്ലോബൽ ഡാറ്റയുടെ സാധ്യതകൾ അതായത് സ്ട്രക്ചർ നിർവചനം `main()` ന്റെ ഉള്ളിലായാൽ ആ സ്ട്രക്ചറിന് `main()` ന്റെ ഉള്ളിൽ മാത്രമേ വേരിയബിളുകൾ നിർമിച്ച് ഉപയോഗിക്കാൻ കഴിയൂ. എന്നാൽ സ്ട്രക്ചർ നിർവചനം `main()` ന് പുറത്താണെങ്കിൽ ആ സ്ട്രക്ചറിനെ ഏത് ഫങ്ഷനിലും വേരിയബിളുകൾ നിർമിക്കാൻ ഉപയോഗിക്കാം.

പ്രോഗ്രാം 1.1 `fflush()` എന്ന ഒരു ഫങ്ഷൻ `gets()` ഉപയോഗിക്കുന്നതിന് മുമ്പായി ഉപയോഗിച്ചിട്ടുണ്ട്. ഇതിനു കാരണം പ്രോഗ്രാമിൽ നമ്മൾ `gets()` ന് ശേഷം മറ്റൊരു ഇൻപുട്ട് നൽകുകയാണെങ്കിൽ, തൊട്ടുമുമ്പ് `gets()` ലെ സ്ട്രിംഗിന്റെ റീഡിങ്ങ് അവസാനിപ്പിക്കാൻ നാം നൽകിയ `'\n'` ക്യാരക്ടർ (ന്യൂലൈൻ) ഇൻപുട്ട് ബഫറിൽ തന്നെ തങ്ങിനിൽക്കും. ഇതായിരിക്കും അടുത്ത ഇൻപുട്ട് ഡാറ്റയായി സ്വീകരിക്കപ്പെടുക. ഈ പ്രശ്നം ഒഴിവാക്കാൻ `fflush()` ഫങ്ഷനിലൂടെ ഇൻപുട്ട് ബഫർ ശൂന്യമാക്കിയ ശേഷം മാത്രമെ അടുത്ത ഇൻപുട്ട് നൽകാൻ പാടുള്ളൂ.

പ്രോഗ്രാം 1.1 ൽ ഒരു സ്ട്രക്ചർ വേരിയബിൾ മാത്രമേ ഉപയോഗിച്ചിട്ടുള്ളൂ. അതിനാൽ പ്രോഗ്രാമിൽ ഒരു കുട്ടിയുടെ മാർക്ക് വിവരങ്ങൾ മാത്രമെ കൈകാര്യം ചെയ്യാൻ കഴിയൂ. കൂടുതൽ കുട്ടികളുടെ വിവരങ്ങൾ കൈകാര്യം ചെയ്യാൻ നമ്മൾ സ്ട്രക്ചർ വേരിയബിളിന്റെ അറെ (array) ഉപയോഗിക്കേണ്ടതായി വരും. നമുക്ക് മറ്റൊരു ഉദാഹരണത്തിലൂടെ സ്ട്രക്ചറുകളുടെ അറെ പരിചയപ്പെടാം.

പ്രോഗ്രാം 1.2 ൽ ഒരു കൂട്ടം വിൽപ്പനക്കാരുടെ വിവരങ്ങൾ സ്വീകരിക്കുന്നു. സെയിൽസ്മാൻ കോഡ്, പേര്, 12 മാസത്തെ സെയിൽസ് ഡാറ്റ എന്നിവയാണ് സ്ട്രക്ചറിലെ അടിസ്ഥാന വിവരങ്ങൾ. ഈ പ്രോഗ്രാം നൽകിയ വിവരങ്ങളുടെ കൂടെ, ഓരോ വിൽപ്പനക്കാരന്റെയും ശരാശരി വിൽപ്പന എത്രയാണെന്നും പ്രദർശിപ്പിക്കുന്നു. ഈ പ്രോഗ്രാമിൽ ഉപയോഗിക്കുന്ന സ്ട്രക്ചറിൽ ഒരു ഫ്ളോട്ടിംഗ് പോയിന്റ് അറെ ഉപയോഗിച്ചിട്ടുണ്ട്.

Program 1.2: വിൽപ്പനക്കാരന്റെ ശരാശരി വിൽപന കാണാൻ

```
#include <iostream>
#include <cstdio>
#include <iomanip>    //setw() എന്ന ഫങ്ഷൻ ഉപയോഗിക്കാൻ
using namespace std;
struct sales_data
{
    int code;
    char name[15];
    float amt[12]; //12 മാസത്തെ വിൽപ്പന സംഖ്യ സൂക്ഷിക്കാൻ ഉള്ള അറെ
    float avg;
};
int main()
{
    sales_data s[20]; //സ്ട്രക്ചറിന്റെ അറെ
    short n,i,j; //short മെമ്മറി ഉപയോഗം കുറയ്ക്കുന്നു
    float sum;
    cout<<"Enter the number of salesmen: ";
    cin>>n;
    for(i=0; i<n; i++)
    {
        cout<<"Enter details of Salesman "<<i+1;
        cout<<"\nSalesman Code: ";
        cin>>s[i].code;
        fflush(stdin);
        cout<<"Name: ";
        gets(s[i].name);
        cout<<"Amount of sales in 12 months: ";
        for(sum=0,j=0; j<12; j++)
```

```
        {
            cin>>s[i].amt[j];
            sum=sum+s[i].amt[j];
        }
        s[i].avg=sum/12;
    }
    cout<<"\t\tDetails of Sales\n";
    cout<<"Code\t\tName\t\tAverage Sales\n";
    for(i=0;i<n;i++)
    {
        cout<<setw(4)<<s[i].code<<setw(15)<<s[i].name;
        for (j=0;j<12;j++)
            cout<<setw(4)<<s[i].amt[j];
        cout<<s[i].avg<<'\n';
    }
    return 0;
}
```

നിങ്ങൾ ഈ പ്രോഗ്രാം ലാബിൽ ചെയ്ത് നോക്കി അതിന്റെ ഔട്ട്പുട്ട് നിരീക്ഷിക്കുക. ഈ പ്രോഗ്രാമിൽ സ്ട്രക്ചർ അംഗമായി ഫ്ളോട്ടിംഗ് പോയിന്റ് അറെ വേരിയബിൾ ഉപയോഗിച്ചിട്ടുണ്ട്. `int` ഡാറ്റാടൈപ്പ് 4 ബൈറ്റ് ഉപയോഗിക്കുന്നതിനാൽ `n`, `i` and `j` എന്നീ വേരിയബിളുകൾ `short` ഡാറ്റ ടൈപ്പുപയോഗിച്ചാണ് ഈ പ്രോഗ്രാമിൽ പ്രഖ്യാപിച്ചിരിക്കുന്നത്.

1.1.4 നെസ്റ്റഡ് സ്ട്രക്ചർ

ഒരു സ്ട്രക്ചറിന്റെ അംഗം മറ്റൊരു സ്ട്രക്ചർ വേരിയബിൾ ആവാം. ഇങ്ങനെയുള്ള സ്ട്രക്ചറിനെ നെസ്റ്റെഡ് സ്ട്രക്ചർ (**nested structure**) എന്ന് വിളിക്കുന്നു. ഈ ആശയം വളരെ ശക്തമായ ഡാറ്റാ സ്ട്രക്ചറുകൾ നിർമിക്കാൻ സഹായകമാണ്. നമ്മൾക്ക് പ്രവേശന തീയതി (Admission date) കൂടി `student` സ്ട്രക്ചറിൽ ഉൾപ്പെടുത്തണമെങ്കിൽ, പട്ടിക 1.2 ൽ നൽകിയിരിക്കുന്ന നിർവചനങ്ങളിൽ (`Definition A`, `Definition B`) ഏതെങ്കിലും ഒന്ന് ഉപയോഗിക്കാം.

Definition A	Definition B

struct date
{
short day;
short month;
short year;
};
struct student
{
int adm_no;
char name[20];
date dt_adm;
float fee;
};
	struct student
	{
	int adm_no;
	char name[20];
	struct date
	{
	short day;
	short month;
	short year;
	} dt_adm;
	float fee;
	};

പട്ടിക 1.2 : രണ്ട് തരം നെസ്റ്റിംഗുകൾ

പട്ടിക 1.2 ലെ ഡെഫനിഷൻ A യിൽ രണ്ട് സ്ട്രക്ചറുകളും വെവ്വേറെ നിർവചിച്ചിട്ടുണ്ട്. student എന്ന സ്ട്രക്ചർ dt_adm എന്ന സ്ട്രക്ചർ വേരിയബിളിനെ അംഗമാക്കിയിട്ടുണ്ട്. ഇത് date തരത്തിലുള്ള സ്ട്രക്ചർ വേരിയബിളാണ്.

അകത്തുള്ള സ്ട്രക്ചർ നെസ്റ്റിംഗ് നടത്തുന്നതിന് മുമ്പ് നിർവചിക്കണം എന്നത് ഇവിടെ നമ്മൾ ഉറപ്പ് വരുത്തേണ്ടതാണ്. എന്നാൽ ഡെഫനിഷൻ - B യിൽ സ്ട്രക്ചർ date തന്നെ സ്ട്രക്ചർ studentന് ഉള്ളിൽ നിർവചിച്ചിരിക്കുന്നു. ഈ തരത്തിൽ ഉപയോഗിച്ചാൽ സ്ട്രക്ചർ date ന്റെ സ്കോപ്പ് സ്ട്രക്ചർ studentന് ഉള്ളിൽ മാത്രമായി ചുരുങ്ങും. അതിനാൽ പുറത്തെവിടെയും date സ്ട്രക്ചർ ടൈപ്പിൽ വേരിയബിളുകൾ പ്രഖ്യാപിക്കാൻ സാധ്യമല്ല. വേണമെങ്കിൽ അകത്തുള്ള സ്ട്രക്ചർ നിർവചനത്തിലെ ടാഗിന്റെ പേര് ഒഴിവാക്കാം. ഒരു നെസ്റ്റഡ് സ്ട്രക്ചർ വേരിയബിളിന് പ്രാരംഭവില നൽകുന്നത് ചുവടെ ചേർക്കുന്നു.

```
student s = {4325, "Vishal", {10, 11, 1997}, 575};
```

```
cout<<s.adm_no<<s.name;
```

```
cout<<s.dt_adm.day<<"/"<<s.dt_adm.month<<"/"<<s.dt_adm.year;
```

ഇന്നർ സ്ട്രക്ചർ എലിമെന്റുകളെ ഉപയോഗിക്കാനുള്ള രീതി ചുവടെ ചേർക്കുന്നു.

```
outer_structure_varaiable.inner_structure_variable.element
```

നമുക്ക് ചെയ്യാം

എംപ്ലോയികോഡ്, പേര്, പ്രവേശനതീയതി, ജോലി, ശമ്പളം എന്നീ വിവരങ്ങൾ അടങ്ങിയ employee എന്ന സ്ട്രക്ചർ നിർമിക്കുക. employee ടൈപ്പിൽ ഉള്ള വേരിയബിളിന്റെ മെമ്മറി ഘടനവരച്ച് അതിന്റെ വലുപ്പം കണ്ടു പിടിക്കുക.

അറെയും സ്ട്രക്ചറും തമ്മിലുള്ള താരതമ്യം

ഒരു പേരിലൂടെ അനേകം ഡാറ്റയെ സൂചിപ്പിക്കാനുപയോഗിക്കുന്ന ഡാറ്റാ തരങ്ങളാണ് സ്ട്രക്ചറും, അറെയും. എന്നാൽ അവ ചില കാര്യങ്ങളിൽ വ്യത്യസ്തമാണ്. അവയെക്കുറിച്ചുള്ള താരതമ്യം ചുവടെ പട്ടികയിൽ നൽകിയിരിക്കുന്നു.

അറെ	സ്ട്രക്ചർ
1. ഇത് ഒരു രൂപീകൃത ഡാറ്റാ ഇനമാണ് (Derived data type)	1. ഇത് ഒരു ഉപയോക്തൃ നിർവചിത ഡാറ്റാഇനമാണ്.
2. ഒരേ ഇനം ഡാറ്റയുടെ കൂട്ടമാണ്.	2. വിവിധതരം ഡാറ്റകളുടെ കൂട്ടമാണ്.
3. അറെയിലെ അംഗങ്ങളെ സൂചിപ്പിക്കുന്നത് സബ്സ്ക്രിറ്റ് ഉപയോഗിച്ചാണ്.	3. സ്ട്രക്ചറിലെ അംഗങ്ങളെ സൂചിപ്പിക്കുന്നത് ഡോട്ട് ഓപ്പറേറ്റർ (.) ഉപയോഗിച്ചാണ്.
4. അറെയിലെ ഒരു അംഗം മറ്റൊരു അറെയെ പ്രതിനിധാനം ചെയ്യുമ്പോൾ മൾട്ടി ഡയമൻഷണൽ അറെ രൂപീകരിക്കപ്പെടുന്നു.	4. സ്ട്രക്ചറിലെ ഒരു അംഗം മറ്റൊരു സ്ട്രക്ചറിനെ പ്രതിനിധാനം ചെയ്യുന്നു എങ്കിൽ അത് നെസ്റ്റെഡ് സ്ട്രക്ചറിനെ രൂപീകരിക്കുന്നു.
5. സ്ട്രക്ചറിന്റെ അറെ നിർമിക്കൽ സാധ്യമാണ്.	5. സ്ട്രക്ചറിലെ അംഗമായി അറെ ഉപയോഗിക്കാം.

പട്ടിക 1.3: അറെയും സ്ട്രക്ചറും തമ്മിലുള്ള താരതമ്യം

നിങ്ങളുടെ പുരോഗതി അറിയാം

1. സ്ട്രക്ചർ എന്നാൽ എന്ത്?
2. സ്ട്രക്ചർ വിവിധതരം ഡാറ്റകളെ ഒരു യൂണിറ്റായി ഏകോപിപ്പിക്കുന്നു - ശരി/തെറ്റ്.
3. സ്ട്രക്ചർ അംഗത്തെ ലഭിക്കുന്നതിന് താഴെ കൊടുത്തവയിൽ ഏതാണ് ശരി.
   a. `struct.element`
   b. `structure_tag.element`
   c. `structure_variable.element`
   d. `structure_tag.structure_variable`
4. നെസ്റ്റഡ് സ്ട്രക്ചർ എന്നാൽ എന്ത്? ഉദാഹരണമെഴുതുക.
5. അറെയ്ക്ക് സബ്സ്ക്രിപ്റ്റ് ആണെങ്കിൽ സ്ട്രക്ചറിന് _____ ആണ്.

1.2 പോയിന്ററുകൾ

നമ്മൾ അഡ്വാൻസ്ഡ് കമ്പ്യൂട്ടിംങ് എന്ന വിഷയത്തിൽ ഒരു അസൈൻമെന്റ് പേപ്പർ തയാറാക്കുകയാണ് എന്ന് കരുതുക. നമുക്ക് അനുയോജ്യമായ പുസ്തകങ്ങൾ നാം പരിശോധിക്കേണ്ടതായി വരും. തീർച്ചയായും നമ്മൾ ലൈബ്രറിയിൽ പരതി നോക്കും. നമ്മൾക്ക് പുസ്തകം കണ്ടു പിടിക്കാൻ കഴിഞ്ഞില്ലെങ്കിൽ ലൈബ്രേറിയനോ, കമ്പ്യൂട്ടർ

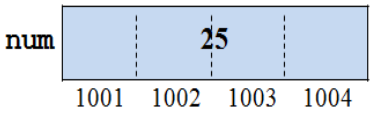
സയൻസ് അധ്യാപകനോ നമ്മളെ ഇക്കാര്യത്തിൽ സഹായിക്കും. ഇവിടെ നമുക്ക് ലൈബ്രേറിയന്റെ/അധ്യാപകന്റെ പങ്കെന്താണെന്ന് പരിശോധിക്കാം. അദ്ദേഹം എല്ലായ്പ്പോഴും ഒരു റഫറൻസാണ്. ലൈബ്രേറിയിലെവിടേയോ ഉള്ള പുസ്തകം കണ്ടെത്തി നമുക്ക് തരാൻ അദ്ദേഹം സഹായിക്കുന്നു.

ചിത്രം 1.4: റഫറൻസിന്റെ ഉദാഹരണം

മുകളിൽ കൊടുത്ത ഉദാഹരണത്തിലെ ലൈബ്രേറിയന്റെ/അധ്യാപകന്റെ പങ്കാണ് പോയിന്റർ വേരിയബിളിനും ഉള്ളത്. പോയിന്റർ കമ്പ്യൂട്ടർ മെമ്മറിയിലേക്ക് ഉള്ള ഒരു റഫറൻസ് ആണ്. താഴെ കൊടുത്ത വാചകം പരിഗണിക്കുക. ചിത്രം 1.4 : റഫറൻസിന്റെ ഉദാഹരണം.

```
int num=25;
```

ഇത് ഒരു പ്രാരംഭവിലനൽകൽ പ്രസ്താവന യാണെന്ന് നമുക്കറിയാം. ഇതിൽ num എന്ന വേരിയബിൾ പ്രഖ്യാപിക്കുന്നതോടൊപ്പം അതിലേക്ക് 25 എന്ന സംഖ്യ ശേഖരിക്കുകയും ചെയ്യുന്നു. ഇവിടെയുള്ള മെമ്മറി നീക്കിവയ്ക്കലിന്റെ ചിത്രം 1.5 ൽ കാണിച്ചിരിക്കുന്നു. ചിത്രത്തിൽ നിന്നും നമുക്ക് വേരിയബിളിന്റെ മൂന്ന് സ്വഭാവ ഘടകങ്ങളെ മനസ്സിലാക്കാം - പേര്, വിലാസം (Address), ഉള്ളടക്കം (Content) എന്നിവയാണ് ആ ഘടകങ്ങൾ. ഇവിടെ വേരിയബിളിന്റെ പേര് num, അതിന്റെ ഉള്ളടക്കം 25. num ഒരു int ടൈപ്പ് വേരിയബിളാണ്. ഇതിനു വേണ്ടി 4 ബൈറ്റ് മെമ്മറിയാണ് ഉപയോഗിക്കപ്പെടുന്നത് (GCCയിൽ).



ചിത്രം 1.5: മെമ്മറി നീക്കിവയ്ക്കൽ

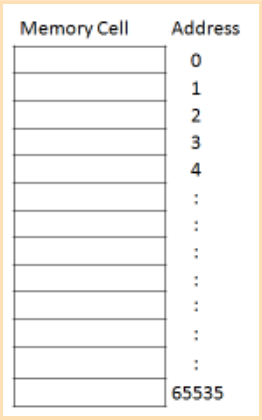
RAM ലെ ഓരോ സെല്ലിലും 1 ബൈറ്റ് മെമ്മറിയാണ് ഉപയോഗിക്കുന്നത്. ഓരോ സെല്ലിനും വ്യത്യസ്തമായ അഡ്രസ്സ് ഉണ്ട്. ഒന്നിലധികം സെല്ലുകൾ ചേർന്ന് ഒരു സ്റ്റോറേജ് ലൊക്കേഷൻ ആയി ഉപയോഗിക്കുമ്പോൾ അതിനെ മെമ്മറി വേഡ് എന്നു പറയുന്നു. 1001, 1002, 1003, 1004 എന്നീ മെമ്മറി സ്ഥാനങ്ങൾ (Locations) വേരിയബിൾ ഉപയോഗിക്കുന്നു. ഇതിൽ തുടക്കത്തിലെ സെല്ലിന്റെ വിലാസമായ (ബേസ് അഡ്രസ്സ്) 1001 ആണ് നമ്മൾ num എന്ന വേരിയബിളിന്റെ വിലാസമായി പരിഗണിക്കുന്നത്. 11-ാം ക്ലാസ്സിൽ നമ്മൾ പഠിച്ച ഒരു കാര്യം ഓർമയുണ്ടല്ലോ, ഒരു വേരിയബിളിന് L വില, R വില എന്നിങ്ങനെ രണ്ട് വിലകൾ ഉണ്ട്. ഇതിൽ L വില വിലാസത്തെയും, R വില ഉള്ളടക്കത്തേയും പ്രതിനിധീകരിക്കുന്നു. ചിത്രം 1.5 ൽ num എന്ന വേരിയബിളിനു L വില 1001 ഉം R വില 25 ഉം ആണ്.

നമുക്ക് ഒരു വേരിയബിളിന്റെ വിലാസം (L-value) മറ്റൊരു വേരിയബിളിൽ ശേഖരിക്കണം എന്ന് കരുതുക. ഇതിനുപയോഗിക്കുന്ന വേരിയബിളിനെ പോയിന്റർ വേരിയബിൾ എന്നു പറയുന്നു. ഇപ്രകാരം നമുക്ക് പോയിന്ററിനെ (Pointer) മെമ്മറി വിലാസം സൂക്ഷിക്കുന്ന ഒരു വേരിയബിളായി നിർവചിക്കാം. പോയിന്റർ ഒരു അടിസ്ഥാന ആശയമാണ്. കാരണം ഇത് മെമ്മറി വിലാസത്തിനെ അതിന്റെ യഥാർഥ രൂപത്തിൽ (Atomic) ഉൾക്കൊള്ളുന്നു. അതുകൊണ്ട് പോയിന്റർ എന്നാൽ മെമ്മറി സ്ഥാനത്തേക്ക്-അതിന്റെ ഉള്ളടക്കത്തിലേക്ക് - ചൂണ്ടുന്ന ഒരു വേരിയബിളായി നമുക്ക് പറയാം.

നമ്മൾ ഒരു വേരിയബിളിന്റെ മെമ്മറി വിലാസം പ്രദർശിപ്പിച്ചു നോക്കിയാൽ ഹെക്സാഡെസിമൽ സംഖ്യകളായ 0x7ffe69878784, 0x7ffe69878785 തുടങ്ങിയ വയായിരിക്കും ലഭിക്കുക. ഇതിനുകാരണം ഇക്കാലത്ത് മെമ്മറി വിലാസം സൂചിപ്പിക്കപ്പെടുന്നത് ഹെക്സാ ഡെസിമൽ നമ്പറുകളിലാണ് എന്നതാണ്. എന്നാൽ ഈ പുസ്തകത്തിൽ സൗകര്യത്തിനായി മെമ്മറി വിലാസങ്ങൾ 1001, 1002 തുടങ്ങിയ ഡെസിമൽ പൂർണസംഖ്യകളുപയോഗിച്ച് രേഖപ്പെടുത്തിയിരിക്കുന്നു (ചിത്രം 1.5 ശ്രദ്ധിക്കുക).

1964 ലെ പോയിന്ററിന്റെ കണ്ടു പിടുത്തം അറിയപ്പെടുന്നത് ഹെറോൾഡ് ലോസൻ (ജനിച്ചത് 1937) എന്ന സോഫ്റ്റ് വെയർ എഞ്ചിനീയറുടെ (ഇദ്ദേഹം ഒരു സിസ്റ്റം എഞ്ചിനീയറും കംപ്യൂട്ടർ ആർക്കിടെക്റ്റുമായിരുന്നു) പേരിലാണ്. 2000 ത്തിൽ, ലോസൻ ഈ കണ്ടുപിടുത്തത്തിന്റെ പേരിൽ IEEE യുടെ കംപ്യൂട്ടർ പയനിയർ അവാർഡ് നേടുകയുണ്ടായി.

പ്രോഗ്രാം നിർദേശങ്ങളും, വേരിയബിളിന്റെ വിലകളും ശേഖരിക്കാനാണ് കംപ്യൂട്ടർ അതിന്റെ മെമ്മറി ഉപയോഗിക്കുന്നത് എന്ന് നിങ്ങൾക്കറിയാം. മെമ്മറി എന്നത് ചിത്രം 1.6 കാണിച്ചതുപോലെ സ്റ്റോറേജ് സെല്ലുകളുടെ തുടർച്ചയായ ഒരു ശേഖരമാണ്, സാധാരണമായി ഓരോ സെല്ലിനും ഒരു ബൈറ്റ് വലുപ്പമാണ് ഉള്ളത്. ഓരോ സെല്ലിനും അതിന്റേതായ മെമ്മറി വിലാസം ഉണ്ടായിരിക്കും. വിലാസങ്ങൾ പൂജ്യത്തിൽ തുടങ്ങി തുടർച്ചയായി നമ്പർ ചെയ്തിരിക്കും. അവസാനത്തെ മെമ്മറിസെല്ലിന്റെ വിലാസം മെമ്മറിയുടെ വലുപ്പത്തെ ആശ്രയിച്ചിരിക്കും. 64 K (64 x 1024 = 65536 Bytes) മെമ്മറി വലുപ്പം ഉള്ള ഒരു കംപ്യൂട്ടറിന്റെ മെമ്മറിയുടെ അവസാനത്തെ വിലാസം 65,535 ആയിരിക്കും.



ചിത്രം 1.6: മെമ്മറിയുടെ ഘടന

നമ്മൾ ഒരു വേരിയബിളിനെ പ്രഖ്യാപിക്കുമ്പോൾ ആ വേരിയബിളിന് വേണ്ടി മെമ്മറിയിലെവിടെയോ ഒരു സ്ഥലം സൃഷ്ടിക്കപ്പെടുന്നു. ഈ സ്ഥലത്താണ് അതിന്റെ R വില (ഉള്ളടക്കം) ശേഖരിക്കപ്പെടുന്നത്. ഓരോ വേരിയബിളിനും അതിന്റേതായ വിലാസം ഉണ്ടായിരിക്കും. ഇക്കാലത്ത് RAMന്റെ വലുപ്പം GBയിലും, മെമ്മറി സ്ഥാനത്തിന്റെ വിലാസം ഹെക്സാഡെസിമൽ നമ്പറിലും ആണ് രേഖപ്പെടുത്തുന്നത്. ഇതിനുകാരണം ഡെസിമൽ നമ്പർസിസ്റ്റത്തെ അപേക്ഷിച്ച് ഹെക്സാഡെസിമൽ നമ്പർസിസ്റ്റത്തിന് കുറഞ്ഞ അക്കങ്ങൾ ഉപയോഗിച്ച് വലിയ സംഖ്യകൾ രേഖപ്പെടുത്തൽ കഴിയും എന്നതാണ്.

1.2.1 പോയിന്റർ വേരിയബിളിന്റെ പ്രഖ്യാപനം (Declaration)

പോയിന്റർ ഒരു രൂപീകൃത ഡാറ്റാതരമാണ് (Derived data type). അതുകൊണ്ടു തന്നെ പ്രോഗ്രാമിൽ ഉപയോഗിക്കുന്നതിന് മുമ്പ് പോയിന്റർ വേരിയബിളുകളെ പ്രഖ്യാപിക്കേണ്ടതായിട്ടുണ്ട്. താഴെ കൊടുത്ത വാക്യഘടനയിലാണ് പോയിന്റർ വേരിയബിൾ പ്രഖ്യാപിക്കുന്നത്.

```
data_type * variable;
```

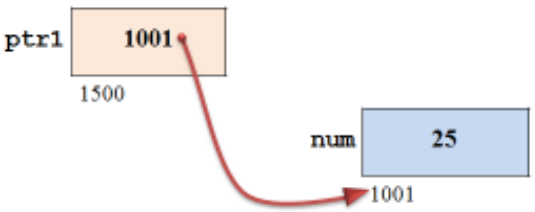
ഇവിടെ ഡാറ്റാതരം എന്നത് അടിസ്ഥാന ഡാറ്റാതരമോ, ഉപയോക്ത്യ നിർവചിത ഡാറ്റാതരമോ ആവാം. variable എന്നാൽ ഐഡന്റിഫയർ ആണ്. ഡാറ്റാതരത്തിനും വേരിയബിളിനും ഇടയിലുള്ള asterisk (*) ചിഹ്നം ശ്രദ്ധിക്കുക. ഇത് പോയിന്റർ വേരിയബിളിന്റെ പ്രഖ്യാപനത്തിലെ പ്രത്യേകതയാണ്. ചുവടെ ചേർത്തവ പോയിന്റർ വേരിയബിൾ പ്രഖ്യാപനത്തിന്റെ ഉദാഹരണങ്ങളാണ്.

```
int *ptr1;
float *ptr2;
struct student *ptr3;
```

സാധാരണ പോലെ പോയിന്റർ വേരിയബിളുകൾക്കും മെമ്മറി നീക്കിവയ്ക്കൽ നടക്കുന്നു. പോയിന്ററിന് ആവശ്യമായ മെമ്മറി സ്ഥാനത്തിന്റെ വലുപ്പം ഡാറ്റാതരത്തിന് അനുസരിച്ചാണെന്നു നിങ്ങൾ വിചാരിക്കുന്നുണ്ടോ? നമുക്കറിയാം മെമ്മറി വിലാസത്തിന്റെ തരം അൺസൈൻഡ് ഇന്റിജർ ആണ്. ഇതിനർഥം എല്ലാ പോയിന്ററും അൺസൈൻഡ് ഇന്റിജർ തരത്തിൽ പ്രഖ്യാപിക്കണം എന്നല്ല. പിന്നെ എന്താണ് പോയിന്ററിന്റെ ഡാറ്റാ തരം തീരുമാനിക്കുന്നതിലെ മാനദണ്ഡം? ഒരു പോയിന്റർ വേരിയബിളിന്റെ ഡാറ്റാതരം അത് പോയിന്റ് ചെയ്യുന്ന ഡാറ്റായുടെ തരം തന്നെയാണ്. മുകളിൽ കൊടുത്ത ഉദാഹരണത്തിൽ ptr1 എന്ന വേരിയബിളിൽ ഒരു ഇന്റിജർ മെമ്മറി സ്ഥാനത്തിന്റെ വിലാസമാണ് സൂക്ഷിക്കപ്പെടുന്നത്. ഇത് പോലെ ptr2 ഒരു ഫ്ളോട്ടിങ് പോയിന്റ് ലൊക്കേഷന്റെയും ptr3 student തരത്തിലുള്ള സ്ട്രക്ചറിന്റെയും വിലാസം ഉൾക്കൊള്ളുന്നു. കൂടുതൽ മെമ്മറി സ്ഥാനങ്ങൾ ഉള്ള ഡാറ്റാതരങ്ങളെ പോയിന്റ് ചെയ്യുമ്പോൾ അതിന്റെ തുടക്കത്തിലെ ലൊക്കേഷന്റെ വിലാസമാണ് (Base Address) പോയിന്റർ വേരിയബിളിൽ ശേഖരിക്കപ്പെടുന്നത്. അപ്പോൾ ഒരു പോയിന്റർ വേരിയബിളിന്റെ മെമ്മറി വലുപ്പം എന്തായിരിക്കും? ഈ വലുപ്പം കമ്പ്യൂട്ടറിലെ വിലാസ സമ്പ്രദായത്തിനനുസരിച്ചായിരിക്കും (Addressing scheme). സാധാരണയായി C++ ൽ ഒരു പോയിന്റർ വേരിയബിളിന്റെ മെമ്മറി വലുപ്പം 2 മുതൽ 4 ബൈറ്റ് വരെ ആയിരിക്കും. ഒരു പ്രോഗ്രാമറെ സംബന്ധിച്ചിടത്തോളം പ്രശ്നങ്ങൾ നിർധാരണം ചെയ്യുമ്പോൾ പോയിന്ററിന്റെ വലുപ്പം അറിയണമെന്ന് നിർബന്ധമില്ല.

1.2.2 &, * എന്നീ ഓപ്പറേറ്ററുകൾ

ഒരിക്കൽ ഒരു പോയിന്റർ വേരിയബിൾ പ്രഖ്യാപിച്ചാൽ അതേ തരത്തിലുള്ള മെമ്മറി ലൊക്കേഷന്റെ വിലാസം മാത്രമേ അതിൽ ശേഖരിക്കാൻ കഴിയൂ. സാധാരണ C++ ൽ ഒരു വേരിയബിളിന്റെ പേര് നൽകിയാൽ അതിന്റെ R വില മാത്രമേ സൂചിപ്പിക്കപ്പെടുന്നുള്ളൂ. അപ്പോൾ എങ്ങനെയാണ് L വില അഥവാ വിലാസം ലഭിക്കുന്നത്? C++ ൽ ഇതിനുവേണ്ടി അഡ്രസ്സ് ഓപ്പറേറ്റർ (&) ഉപയോഗിക്കുന്നു. ഉദാഹരണമായി num ഒരു ഇന്റിജർ വേരിയബിളാണെങ്കിൽ അതിന്റെ വിലാസം താഴെ കൊടുത്ത രീതിയിൽ ptr1 എന്ന പോയിന്ററിൽ ശേഖരിക്കാം.



ചിത്രം 1.7 പോയിന്ററും അത് ചൂണ്ടുന്ന ലൊക്കേഷനും

```
ptr1 = &num;
```

പ്രോഗ്രാം പ്രവർത്തിപ്പിക്കുന്ന സമയത്ത് ഈ വാചകം ptr1 എന്ന പോയിന്റർ വേരിയബിളിനും num എന്ന മെമ്മറി സ്ഥാനത്തിനും ഇടയിൽ ഒരു ബന്ധം സ്ഥാപിക്കുന്നു. ഇത് ചിത്രം 1.7 ൽ കാണിച്ചിരിക്കുന്നു.

പോയിന്റർ ഒരു റഫറൻസ് ആണെന്നു നാം മനസ്സിലാക്കി. അതുകൊണ്ട് മെമ്മറിയിൽ എവിടെ ഡാറ്റ ശേഖരിച്ചാലും അത് സൂചിപ്പിക്കാൻ (refer) പോയിന്റർ വേരിയബിളിനു കഴിയും. പോയിന്റർ വേരിയബിളിന്റെ ഡീറഫെറൻസിലൂടെ നമുക്ക് ഡാറ്റ ലഭ്യമാവും. ഇതിനുവേണ്ടി ഇൻഡയറക്ഷൻ ഓപ്പറേറ്റർ (indirection operator) അഥവാ (*) ഉപയോഗിക്കുന്നു. ഇതിനെ ഡീറഫറൻസ് ഓപ്പറേറ്റർ (dereference operator) എന്നും പറയുന്നു. താഴെ കൊടുത്ത C++ പ്രസ്താവന ptrl എന്ന പോയിന്റർ വേരിയബിൾ ചൂണ്ടുന്ന ലൊക്കേഷനിലെ വില പ്രദർശിപ്പിക്കുന്നു.

```
cout << *ptr1;
```

ഈ പ്രസ്താവന cout << num; എന്ന പ്രസ്താവനയ്ക്ക് തുല്യമാണെന്ന് വ്യക്തമാണല്ലോ.

* ഓപ്പറേറ്റർ ഒരു പോയിന്റർ ലൊക്കേഷനിലെ വില തിരിച്ച് തരുന്നതു കൊണ്ട് 'വാല്യൂ അറ്റ് ഓപ്പറേറ്റർ' എന്നും അറിയപ്പെടുന്നു.
& ഓപ്പറേറ്ററും * ഓപ്പറേറ്ററും യൂനറി ഓപ്പറേറ്ററുകളുമാണ്. എല്ലാ വേരിയബിളിനും മെമ്മറി സ്ഥാനവും വിലാസവും ഉള്ളതിനാൽ & ഓപ്പറേറ്റർ എല്ലാ വേരിയബിളിന്റെയും കൂടെ ഉപയോഗിക്കാം എന്നാൽ * ഓപ്പറേറ്റർ പോയിന്റർ വേരിയബിളിന്റെ കൂടെ മാത്രമേ ഉപയോഗിക്കാൻ കഴിയുകയുള്ളൂ.

ചിത്രം 1.7 ൽ കൊടുത്ത വേരിയബിൾ പരിശോധിക്കൂ. ചുവടെ കൊടുത്ത ഉദാഹരണങ്ങൾ &, * ഓപ്പറേറ്ററുകളുടെ ഉപയോഗം കാണിക്കുന്നു.

```
cout<< &num;  // 1001 num ന്റെ വിലാസം പ്രദർശിപ്പിക്കുന്നു
cout<< ptr1;  // 1001 ptr1 ന്റെ ഉള്ളടക്കം പ്രദർശിപ്പിക്കുന്നു
cout<< num;   // 25 num ന്റെ ഉള്ളടക്കം പ്രദർശിപ്പിക്കുന്നു
cout<< *ptr1;/* 25 ptr1 പോയിന്റ് ചെയ്യുന്ന സ്ഥാനത്തെ വില പ്രദർശിപ്പിക്കുന്നു
cout<< &ptr1;// 1500 (ptr1 ന്റെ വിലാസം പ്രദർശിപ്പിക്കുന്നു.
cout<< *num; // Error!! num എന്നത് പോയിന്റർ വേരിയബിളല്ല.
```

അവസാനത്തെ പ്രസ്താവന തെറ്റാണ്. കാരണം num ഒരു പോയിന്റർ വേരിയബിളല്ല. അതിലുള്ള 25 എന്ന വില ഒരു മെമ്മറിവിലാസവും അല്ല. * ഓപ്പറേറ്റർ പോയിന്റർ വേരിയബിളിനോടു കൂടി മാത്രമേ ഉപയോഗിക്കാൻ പാടുള്ളൂ.

1.3 മെമ്മറി നീക്കിവയ്ക്കലിന്റെ (Allocation) രീതികൾ

വേരിയബിൾ ഡിക്ലറേഷൻ സ്റ്റേറ്റ്മെന്റിലൂടെയാണ്, മെമ്മറി അലോക്കേഷൻ തുടങ്ങുന്നത് എന്ന് നമുക്കറിയാം. പ്രോഗ്രാം റാമിൽ ലോഡ് ചെയ്യുമ്പോൾ ആവശ്യമുള്ള മെമ്മറിയും നീക്കിവയ്ക്കപ്പെടുന്നു. മെമ്മറി നീക്കിവയ്ക്കലിന് ശേഷമാണ് പ്രോഗ്രാം പ്രവർത്തിപ്പിക്കൽ ആരംഭിക്കുന്നത്. മെമ്മറി നീക്കി വയ്ക്കുന്നതിന്റെ അളവ് പ്രോഗ്രാമിൽ ഉപയോഗിച്ചിരിക്കുന്ന വേരിയബിളുകളുടെ എണ്ണത്തെയും തരത്തെയും അനുസരിച്ചാണ്. ഈ അളവ് സ്ഥിരം (static) ആണ്. ഇത് ഒരിക്കലും പ്രോഗ്രാം പ്രവർത്തിപ്പിക്കുന്ന സമയത്ത്

കൂടുകയോ കുറയുകയോ ഇല്ല. ഇങ്ങനെ പ്രോഗ്രാം പ്രവർത്തിപ്പിക്കുന്നതിന് മുമ്പ് നടക്കുന്ന മെമ്മറി നീക്കിവയ്ക്കലിനെ സ്റ്റാറ്റിക് മെമ്മറി അലോക്കേഷൻ (Static Memory Allocation) എന്ന് പറയുന്നു. ഈ നീക്കിവയ്ക്കൽ നടക്കുന്നത് പ്രോഗ്രാമിലെ വേരിയബിൾ പ്രഖ്യാപന പ്രസ്താവനയ്ക്കനുസരിച്ചാണ്. ഇനി മറ്റൊരു തരം മെമ്മറി അലോക്കേഷൻ ആയ ഡൈനാമിക് മെമ്മറി അലോക്കേഷൻ (Dynamic Memory Allocation) പരിചയപ്പെടാം. ഇവിടെ മെമ്മറി നീക്കിവയ്ക്കൽ നടക്കുന്നത് പ്രോഗ്രാം പ്രവർത്തിക്കുമ്പോഴാണ്. ഈ നീക്കിവയ്ക്കൽ നടക്കുന്നത് new എന്ന ഒരു ഓപ്പറേറ്റർ മുഖേനയാണ്. ഇങ്ങനെ നീക്കിവച്ച് മെമ്മറി സ്വതന്ത്രമാക്കാൻ (Deallocate) delete എന്ന ഓപ്പറേറ്റർ ഉപയോഗിക്കുന്നു.

1.3.1 ഡൈനാമിക് ഓപ്പറേറ്ററുകൾ - new, delete

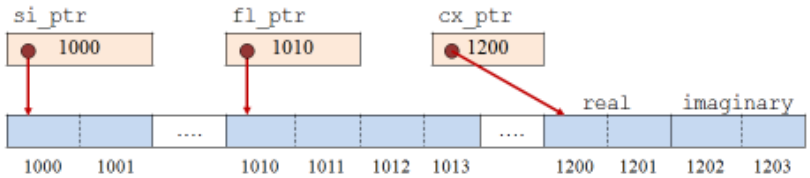
C++ ലെ new എന്ന കീവേഡ് ഒരു ഓപ്പറേറ്റർ ആണ്. ഇത് ഡൈനാമിക് മെമ്മറി അലോക്കേഷനുവേണ്ടി ഉപയോഗിക്കുന്നു. ഈ നീക്കിവയ്ക്കൽ നടക്കുന്നത് പ്രോഗ്രാം പ്രവർത്തിപ്പിക്കുന്ന സമയത്താണ് (റൺ ടൈമിലാണ്). ഇത് ഒരു യൂനറി ഓപ്പറേറ്റർ ആണ്. അടിസ്ഥാന ഡാറ്റാ തരത്തിലെയോ ഉപയോക്തൃനിർവചിത ഡാറ്റാ തരത്തിലേയോ ഒരു ഓപ്പറേറ്ററാണ് ഇതിൽ ഉപയോഗിക്കുന്നത്. new എന്ന ഓപ്പറേറ്ററും ഈ ഓപ്പറേറ്ററും കൂടിച്ചേർന്ന് ഒരു പദപ്രയോഗം (Expression) ഉണ്ടാക്കുകയും. ആ പദപ്രയോഗം ഒരു വില തിരിച്ച് നൽകുകയും ചെയ്യുന്നു. ഒരു മെമ്മറി സ്ഥാനത്തിന്റെ വിലാസമാണ് ആ വില. മെമ്മറി സ്ഥാനത്തിന്റെ വലുപ്പം ഓപ്പറേറ്ററിന്റെ ഡാറ്റാ തരത്തിനനുസരിച്ചായിരിക്കും. ഡൈനാമിക് അലോക്കേഷന് താഴെ കൊടുത്ത വാക്യ ഘടന ഉപയോഗിക്കുന്നു.

```
pointer_variable = new data_type;
```

പോയിന്റർ വേരിയബിൾ ഉപയോഗിക്കുന്നത് new ഓപ്പറേറ്റർ തിരിച്ച് തരുന്ന മെമ്മറി വിലാസം സൂക്ഷിക്കാൻ വേണ്ടിയാണ്. അതുകൊണ്ട് new ഓപ്പറേറ്ററിന് ശേഷം നൽകിയ ഡാറ്റാതരത്തിൽ തന്നെ പോയിന്റർ വേരിയബിൾ നേരത്തേ പ്രഖ്യാപിച്ചിരിക്കണം.

```
short * si_ptr;
float * fl_ptr;
struct complex * cx_ptr;
si_ptr = new short;
fl_ptr = new float;
cx_ptr = new complex;
```

ഇവയുടെ മെമ്മറി നീക്കിവയ്ക്കൽ ചിത്രം 1.8 ൽ കാണിച്ചിരിക്കുന്നു.



ചിത്രം 1.8: ഡൈനാമിക് മെമ്മറി അലോക്കേഷന്റെ ഘടന

ചിത്രം 1.8 ൽ കാണിച്ചിരിക്കുന്നത് ശ്രദ്ധിക്കുക. ഇവിടെ 2 ബൈറ്റ് മെമ്മറി സ്ഥാനം short ടൈപ്പ് ഡാറ്റയ്ക്ക് നീക്കിവയ്ക്കുന്നു. ഇതിന്റെ വിലാസം 1000 ആണ്, ഇതുപോലെ 1010 എന്ന വിലാസത്തിൽ 4 ബൈറ്റ് മെമ്മറി float തരം ഡാറ്റയ്ക്ക് നീക്കിവയ്ക്കുന്നു.

ഇത് സൂക്ഷിക്കുന്നത് fl_ptr എന്ന പോയിന്റർ വേരിയബിളിലാണ്. മുമ്പ് നാം പഠിച്ച complex എന്ന സ്ട്രക്ചർ 2 short തരത്തിലുള്ള അംഗങ്ങളെ ഉൾക്കൊള്ളുന്നു. അതുകൊണ്ട് 1200 എന്ന വിലാസത്തിൽ തുടങ്ങുന്ന 4 ബൈറ്റ് മെമ്മറി ലൊക്കേഷൻ cx_ptr എന്ന complex സ്ട്രക്ചർ തരത്തിലുള്ള പോയിന്റർ വേരിയബിൾ സൂചിപ്പിക്കുന്നു. (short real നും short imaginary യ്ക്കും 2 ബൈറ്റ് വീതം ആകെ 4 ബൈറ്റ്). ഡൈനാമിക് മെമ്മറി അലോക്കേഷൻ നടത്തിയ സ്ഥലങ്ങൾ സാധാരണ വേരിയബിളുകൾക്ക് സൂചിപ്പിക്കാൻ കഴിയില്ല. ഇത് സൂചിപ്പിക്കാൻ പോയിന്റ് വേരിയബിൾ തന്നെ ഉപയോഗിക്കണം. താഴെ കൊടുത്ത ഉദാഹരണം ശ്രദ്ധിക്കുക.

```
*si_ptr = 247;
cin >> *fl_ptr;
```

നമുക്ക് സ്ട്രക്ചർ പോയിന്റർ വേരിയബിളായ cx_ptr ഉണ്ട്. ഇത് സൂചിപ്പിക്കുന്ന ഡാറ്റ മുകളിൽ കൊടുത്ത രീതിയിൽ ഉപയോഗിക്കാൻ കഴിയില്ല. ഇതിന്റെ ഉപയോഗരീതി ഈ അധ്യായത്തിൽത്തന്നെ നമുക്ക് പിന്നീട് മനസ്സിലാക്കാം.

നമുക്ക് സ്റ്റാറ്റിക് മെമ്മറി അലോക്കേഷൻ വേരിയബിളുകൾക്ക് പ്രാരംഭവില നൽകുന്നതുപോലെ ഡൈനാമിക് മെമ്മറി അലോക്കേഷനിലും വേരിയബിളുകൾക്ക് പ്രാരംഭവില നൽകാം. ഇതിന് ചുവടെ ചേർത്ത വാക്യഘടന ഉപയോഗിക്കുന്നു.

```
pointer_variable = new data_type(value);
```

ചുവടെ കൊടുത്തവ ഡൈനാമിക് മെമ്മറി അലോക്കേഷന്റെ കൂടെയുള്ള പ്രാരംഭവില നൽകലിന് ഉദാഹരണങ്ങളാണ്.

```
si_ptr = new short(0);
fl_ptr = new float(3.14);
```

cx_ptr ന്റെ കാര്യത്തിൽ ഈ തരത്തിലുള്ള പ്രാരംഭവില നൽകൽ സാധ്യമല്ല.
ഒരിക്കൽ നമ്മൾ new ഓപ്പറേറ്റർ ഉപയോഗിച്ച് ഡൈനാമിക് മെമ്മറി അലോക്കേഷൻ നടത്തിയാൽ അത് സ്വതന്ത്രമാക്കേണ്ടതായിട്ടുണ്ട് അഥവാ ഡീ അലോക്കേറ്റ് ചെയ്യേണ്ടതായിട്ടുണ്ട്. C++ ൽ ഇതിനുവേണ്ടി delete ഓപ്പറേറ്ററിനെ ഉപയോഗിക്കുന്നു. സ്റ്റാറ്റിക് മെമ്മറി അലോക്കേഷന്റെ കാര്യത്തിൽ വേരിയബിളിന്റെ സ്കോപ്പും, ലൈഫ് ടൈമും നോക്കി ഓപ്പറേറ്റിംഗ് സിസ്റ്റം തന്നെ മെമ്മറി നീക്കിവയ്ക്കലും, സ്വതന്ത്രമാക്കലും നടത്തും. എന്നാൽ ഡൈനാമിക് മെമ്മറി അലോക്കേഷനുമായി ബന്ധപ്പെട്ട മെമ്മറി സ്വതന്ത്രമാക്കാൻ പ്രോഗ്രാമിൽ തന്നെ delete ഓപ്പറേറ്റർ ഉപയോഗിച്ചുള്ള നിർദേശങ്ങൾ നൽകേണ്ടതാണ്. delete ഓപ്പറേറ്ററിന്റെ ഉപയോഗം താഴെ നൽകിയിരിക്കുന്നു.

```
delete pointer_variable;
```

ചുവടെ ഉദാഹരണങ്ങൾ ചേർക്കുന്നു.

```
delete si_ptr;
delete fl_ptr, cx_ptr;
```

1.3.2 മെമ്മറി ലീക്ക് (Memory leak)

new ഓപ്പറേറ്റർ ഉപയോഗിച്ച് നീക്കി വച്ച മെമ്മറി ബ്ലോക്ക് delete ഓപ്പറേറ്റർ ഉപയോഗിച്ച് സ്വതന്ത്രമാക്കുന്നില്ല എങ്കിൽ അതിനെ ഓർഫൻറ് ബ്ലോക്ക് എന്ന് പറയുന്നു. ഇത് ഉപയോഗിക്കാതെ ബാക്കി വരുന്ന മെമ്മറി ബ്ലോക്ക് ആണ്. പക്ഷെ ഇത് വീണ്ടും

ഡാറ്റ സൂക്ഷിക്കാൻവേണ്ടി നീക്കിവയ്ക്കാൻ കഴിയില്ല. പ്രോഗ്രാമിന്റെ ഓരോ പ്രവർത്തിപ്പിക്കലിലും ഇത്തരത്തിലുള്ള ബ്ലോക്കുകൾ സൃഷ്ടിക്കപ്പെടുകയും മെമ്മറിയുടെ ഉപയോഗിക്കത്തക്ക ഭാഗം തുടർച്ചയായി കുറയുകയും ചെയ്യും. ഇങ്ങനെ മെമ്മറി നഷ്ടപ്പെടുന്നതിനെ മെമ്മറി ലീക്ക് എന്ന് പറയുന്നു.

താഴെ പറയുന്നവയാണ് മെമ്മറി ലീക്കിന്റെ കാരണങ്ങൾ.

- ഡൈനാമിക്ക് അലോക്കേഷൻ നടത്തിയ മെമ്മറി (new ഓപ്പറേറ്റർ ഉപയോഗിച്ച്) സ്വതന്ത്രമാക്കാൻ മറന്നു പോകുന്നത്.
- പ്രോഗ്രാമിങ്ങിലെ ലോജിക്കൽ പിഴവ് മൂലം 'delete' നിർദേശം പ്രവർത്തിപ്പിക്കാത്തത്.
- നിലവിൽ ഒരു മെമ്മറി ബ്ലോക്കിനെ പോയിന്റ് ചെയ്യുന്ന പോയിന്ററിലേയ്ക്ക് new ഓപ്പറേറ്ററിലൂടെ ഒരു പുതിയ മെമ്മറി വിലാസം ശേഖരിക്കുന്നത്.

new ഓപ്പറേറ്റർ ഉപയോഗിച്ച് നടത്തിയ മെമ്മറി അലോക്കേഷൻ delete ഓപ്പറേറ്റർ ഉപയോഗിച്ച് സ്വതന്ത്രമാക്കുക എന്നതാണ് മെമ്മറി ലീക്കിനുള്ള പരിഹാരം. ഡൈനാമിക് അലോക്കേഷന്റെ കാര്യത്തിൽ മാത്രമാണ് മെമ്മറി ലീക്ക് ഉണ്ടാകുന്നത്. സ്റ്റാറ്റിക് അലോക്കേഷനിൽ മെമ്മറി നീക്കിവയ്ക്കലും സ്വതന്ത്രമാക്കലും ഓപ്പറേറ്റിംങ് സിസ്റ്റം നേരിട്ട് നടപ്പിലാക്കുന്നു. ഇതിന് പ്രത്യേക നിർദേശത്തിന്റെ ആവശ്യം ഇല്ല. അതുകൊണ്ട് സ്റ്റാറ്റിക് മെമ്മറി അലോക്കേഷനിൽ മെമ്മറി ലീക്കിനുള്ള സാധ്യതയില്ല.

നമുക്ക് ചെയ്യാം

ഇപ്പോൾ നമുക്ക് സ്റ്റാറ്റിക് മെമ്മറി അലോക്കേഷനും ഡൈനാമിക് മെമ്മറി അലോക്കേഷനും താരതമ്യം ചെയ്യാം. പട്ടിക 1.4 ൽ ഈ താരതമ്യം കൊടുത്തിരിക്കുന്നു. ചില ഭാഗങ്ങൾ നിങ്ങൾക്ക് പൂരിപ്പിക്കാൻ വേണ്ടി വിട്ടിരിക്കുന്നു.

സ്റ്റാറ്റിക് മെമ്മറി അലോക്കേഷൻ	ഡൈനാമിക് മെമ്മറി അലോക്കേഷൻ
i. പ്രോഗ്രാം പ്രവർത്തിപ്പിക്കുന്നതിന് മുമ്പ് നടക്കുന്നു.	
ii.	new ഓപ്പറേറ്റർ ആവശ്യമുണ്ട്.
iii.	പോയിന്റർ നിർബന്ധമാണ്.
iv. ഡാറ്റയെ വേരിയബിളുകൾ ഉപയോഗിച്ച് സൂചിപ്പിക്കുന്നു.	
v. ഡീ-അലോക്കേഷന് വേണ്ടി പ്രത്യേകം സ്റ്റേറ്റ്മെന്റ് ആവശ്യമില്ല.	

പട്ടിക 1.4: സ്റ്റാറ്റിക് അലോക്കേഷനും ഡൈനാമിക് അലോക്കേഷനും തമ്മിലുള്ള താരതമ്യം

നിങ്ങളുടെ പുരോഗതി അറിയാം

1. പോയിന്റർ എന്നാൽ എന്ത്?
2. ഒരു പോയിന്ററിന്റെ ഡാറ്റാതരം നിർണയിക്കുന്നതിന്റെ മാനദണ്ഡമെന്ത്?
3. `mks` ഒരു ഇന്റിജർ വേരിയബിളാണെങ്കിൽ, ഈ വേരിയബിളിന്റെ വിലാസം ഒരു പോയിന്റർ വേരിയബിളിൽ ശേഖരിക്കാനുള്ള C++ വാചകം എഴുതുക.
4. `ptr` ഒരു പോയിന്റർ വേരിയബിൾ ആണ്. ഇതുപയോഗിച്ച് ഒരു ഇന്റിജറിന് വേണ്ടി മെമ്മറി നീക്കി വച്ച് അതിൽ 12 എന്ന സംഖ്യ പ്രാരംഭവിലയായ് ചേർക്കുന്നതിനുള്ള C++ വാചകം എഴുതുക.
5. ചുവടെ ചേർത്ത വാചകങ്ങൾ പരിഗണിക്കുക: `int *p, a=5; p=&a; cout<<*p+a;` ഇതിന്റെ ഔട്ട്പുട്ട് എന്താണ്?

1.4 പോയിന്ററിലെ ഓപ്പറേഷനുകൾ

പോയിന്ററുമായി ബന്ധപ്പെട്ട് ഇൻഡയറക്ഷൻ (*) ഓപ്പറേറ്റർ, അഡ്രസ്സ് ഓഫ് (&) ഓപ്പറേറ്റർ എന്നിവ നമ്മൾ ചർച്ച ചെയ്ത് കഴിഞ്ഞല്ലോ. 11-ാം ക്ലാസ്സിൽ അരിത്തമാറ്റിക്, റിലേഷണൽ, ലോജിക്കൽ എന്നീ ഓപ്പറേറ്ററുകളും പരിചയപ്പെട്ടു. ഈ ഭാഗത്ത് പോയിന്റർ വേരിയബിളിൽ ഉപയോഗിക്കുന്ന ഓപ്പറേഷനുകളാണ് നാം പരിചയപ്പെടാൻ പോകുന്നത്.

1.4.1 പോയിന്ററിലെ അരിത്തമാറ്റിക് ഓപ്പറേഷനുകൾ

മെമ്മറി വിലാസം എന്നത് ഒരു സംഖ്യയാണെന്ന് നമുക്ക് അറിയാം. അതിനാൽ ചില അരിത്തമാറ്റിക് ഓപ്പറേഷനുകളും നമുക്ക് പോയിന്റർ വേരിയബിളിൽ പ്രയോഗിക്കാം. ഭാഗം 1.3.1 ൽ ഉപയോഗിച്ചു `si_ptr`, `fl_ptr` എന്നീ പോയിന്റർ വേരിയബിളുകൾ പരിഗണിക്കുക. (ചിത്രം 1.8 കാണുക). ഇനി ചുവടെ ചേർത്ത പ്രസ്താവനകൾ പരിശോധിക്കുക.

```
cout << si_ptr + 1;
cout << fl_ptr + 1;
```

എന്തായിരിക്കും ഔട്ട്പുട്ട്? 1001 ഉം 1011 ആണ് ഔട്ട്പുട്ട് എന്ന് നിങ്ങൾ കരുതുന്നുണ്ടോ.

പോയിന്ററിനോട് 1 കൂട്ടുന്നത് `int` തരത്തിലുള്ള വേരിയബിളിനോടോ `float` തരത്തിലുള്ള വേരിയബിളിനോടോ 1 കൂട്ടുന്നത് പോലെ അല്ല. നമ്മൾ ഒരു `short int` പോയിന്ററിനോട് 1 കൂട്ടുമ്പോൾ, ആ പ്രയോഗം തൊട്ടടുത്ത മെമ്മറി സ്ഥാനത്തിന്റെ വിലാസമായി മാറുന്നു. അതായത് 1000, 1001 എന്നീ ലൊക്കേഷനുകൾ `short int` വേരിയബിൾ ഉപയോഗിക്കുന്ന ലൊക്കേഷനുകളാണ്. 1002 എന്നത് അടുത്ത `short int` ന്റെ ബേസ് അഡ്രസ്സായിരിക്കും. അതായത് നമ്മൾ `short int` പോയിന്റർ വേരിയബിളിനോട് ഒന്ന് കൂട്ടുമ്പോൾ `short int` ന്റെ യഥാർഥ വലുപ്പം (2) ആണ് മെമ്മറി അഡ്രസ്സിനോട് കൂട്ടുന്നത്. ഇതുപോലെ ഒരു ഫ്ളോട്ട് പോയിന്റർ വേരിയബിളിനോട് 1 കൂട്ടുമ്പോൾ അതിന്റെ വലുപ്പമായ 4 ആണ് വിലാസത്തോട് കൂട്ടുന്നത്. അതുകൊണ്ട് `fl_ptr+1`

എന്ന പദപ്രയോഗം നൽകുന്നത് 1014 എന്ന വിലാസമാണ്. അതുകൊണ്ട് si_ptr+4 എന്ന പദപ്രയോഗം 1008 (1000+4×2) എന്ന വിലാസം നൽകുമെന്ന് വ്യക്തമാണ്. ഇതു പോലെ fl_ptr+3 നൽകുന്നത് 1022 (1010+3×4) എന്ന വിലാസമാണ്. ഇതുപോലെ പോയിന്ററർ വേരിയബിളുകളിൽ വ്യവകലനവും നടത്താം. മറ്റ് അരിത്തമാറ്റിക് ഓപ്പറേറ്റ ഷനുകൾ ഒന്നും തന്നെ പോയിന്ററർ വേരിയബിളിനു മുകളിൽ പ്രവർത്തിക്കുകയില്ല. ഇത്തരത്തിലുള്ള ഓപ്പറേഷനുകൾ പ്രായോഗികമായി ചിലപ്പോൾ തെറ്റായേക്കാം. കാരണം

നമുക്ക് ചെയ്യാം

ചുവടെ കൊടുത്തിരിക്കുന്ന അരിത്തമാറ്റിക് പദപ്രയോഗങ്ങൾ നൽകുന്ന വിലകൾ കണ്ടുപിടിക്കുക.

a) si_ptr + 10 b) fl_ptr + 7

c) si_ptr - 5 d) fl_ptr - 10

ഈ ഓപ്പറേഷനിലൂടെ സൂചിപ്പിക്കുന്ന മെമ്മറി സ്ഥാനങ്ങൾ ആക്സസ്റൈറ്റ് പ്രകാരം ഉപയോഗിക്കാൻ അനുവാദമില്ലാത്തവയായിരിക്കും.

ചുവടെ ചേർത്തിരിക്കുന്ന ഉദാഹരണങ്ങൾ പോയിന്ററർ ഓപ്പറേഷനുകൾ വിശദീകരി ക്കുന്നു.

```
int *ptr1, *ptr2;      // രണ്ട് പോയിന്ററർ വേരിയബിളുകളുടെ പ്രഖ്യാപനം
ptr1 = new int(5);     /* ഡൈനാമിക് മെമ്മറി അലോക്കേഷൻ (വിലാസം 1000
                          ആണെന്നിരിക്കട്ടെ), 5 എന്ന സംഖ്യ പ്രാരംഭവിലയായ്
                          ചേർന്നിരിക്കുന്നു. 5*/
ptr2 = ptr1 + 1;       /* ptr2 തൊട്ടടുത്ത ഇന്റിജർ സ്ഥാനമായ 1004 ലേക്ക്
                          പോയിന്റ് ചെയ്യും */
++ptr2;                // ptr2 = ptr2 + 1 എന്ന ഓപ്പറേഷന് തുല്യം
cout<< ptr1;           // 1000 പ്രദർശിപ്പിക്കുന്നു
cout<< *ptr1;          // 5 പ്രദർശിപ്പിക്കുന്നു
cout<< ptr2;           // 1004 പ്രദർശിപ്പിക്കുന്നു
cin>> *ptr2;           /* ഒരു ഇന്റിജർ റീഡ് ചെയ്ത് (12 ആണെന്നിരിക്കട്ടെ)
                          ഇതിനെ 1004 എന്ന ലൊക്കേഷനിൽ ശേഖരിക്കുന്നു */
cout<< *ptr1 + 1;      // 6 (5 + 1) പ്രദർശിപ്പിക്കുന്നു
cout<< *(ptr1 + 2);    // 12 പ്രദർശിപ്പിക്കുന്നു, (1004 ലെ വില)
ptr1--;                // ptr1 = ptr1 - 1 എന്ന പ്രസ്താവനയ്ക്ക്
                          തുല്യം
```

ഇനി നമുക്ക് പോയിന്ററിന്റെ ഓപ്പറേഷനുകൾ വിവരിക്കുന്ന ഒരു പ്രോഗ്രാം എഴുതാം. പ്രോഗ്രാം 1.3 ഒരു കൂട്ടം കുട്ടികളുടെ ശരാശരി ഉയരം കാണാനുള്ളതാണ്.

Program 1.3: കുട്ടികളുടെ ശരാശരി ഉയരം കാണാൻ

```
#include <iostream>
using namespace std;
int main()
{
    int *ht_ptr, n, s=0;
    float avg_ht;
    ht_ptr = new int;        //ഡൈനാമിക് മെമ്മറി അലോക്കേഷൻ
    cout<<"Enter the number of students: ";
    cin>>n;
    for (int i=0; i<n; i++)
    {
        cout<<"Enter the height of student "<<i+1<<" - ";
        cin>>*(ht_ptr+i);//to get the address of the next location
        s = s + *(ht_ptr+i);
    }
    avg_ht = (float)s/n;
    cout<<"Average height of students in the class = "<<avg_ht;
    return 0;
}
```

പ്രോഗ്രാം 1.3 ൽ ഡൈനാമിക് അലോക്കേഷനിലൂടെ ഒരു ഇന്റിജർ മെമ്മറിസ്ഥാനം നിർമിച്ച് അതിന്റെ വിലാസം ht_ptr എന്ന പോയിന്ററർ വേരിയബിളിൽ ശേഖരിക്കുന്നു. ലൂപ്പിന്റെ ബോഡി ആദ്യത്തെ പ്രാവശ്യം പ്രവർത്തിക്കുമ്പോൾ 0 ആണ് വിലാസത്തോട് കൂട്ടുന്നത്. ഇത് പ്രത്യേകിച്ച് മാറ്റമൊന്നും വരുത്തുന്നില്ല. ഇൻപുട്ട് ഡാറ്റ മെമ്മറി സ്ഥാനത്ത് സൂക്ഷിക്കുന്നു. ലൂപ്പ് ബോഡിയുടെ അടുത്ത പ്രവർത്തനത്തിൽ വിലാസത്തോട് 1 കൂട്ടുന്നു. അങ്ങനെ അടുത്ത ഇന്റിജർ സ്ഥാനത്തിൽ രണ്ടാമത്തെ ഡാറ്റ ശേഖരിക്കപ്പെടുന്നു. ഈ പ്രവൃത്തി n തവണ ആവർത്തിക്കപ്പെടുന്നു. അങ്ങനെ n തവണ കുട്ടികളുടെ ഡാറ്റയും തുടർച്ചയായ ഇന്റിജർ സ്ഥാനങ്ങളിൽ ശേഖരിക്കപ്പെടുന്നു. ഓരോ ഡാറ്റയും നൽകുന്ന സമയത്തു തന്നെ അതിന്റെ തുകയും കണക്കാക്കപ്പെടുന്നു. അവസാനം ശരാശരി ഉയരം കണക്കാക്കി പ്രദർശിപ്പിക്കുന്നു. കൃത്യമായ ശരാശരി ഉയരം ലഭിക്കുവാൻ തുകയെ ബാഹ്യതരം മാറ്റലിലൂടെ (Explicit type conversion) ഫ്ളോട്ടാക്കി മാറ്റിയിരിക്കുന്നു. പ്രോഗ്രാമിന്റെ ഔട്ട്പുട്ട് ചുവടെ ചേർക്കുന്നു.

```
Enter the number of students: 5
Enter the height of student 1 - 170
Enter the height of student 2 - 169
Enter the height of student 3 - 175
Enter the height of student 4 - 165
Enter the height of student 5 - 177
Average height of students in the class = 171.199997
```

പ്രോഗ്രാം 1.3 മറ്റൊരു കാര്യം കൂടി വിവരിക്കുന്നു. ഒരേതരം ഡാറ്റയുടെ കൂട്ടത്തെ അനായാസമായി പോയിന്ററിലൂടെ കൈകാര്യം ചെയ്യാം. നമുക്കറിയാം 11-ാം ക്ലാസിൽ ഇതേ

ആവശ്യത്തിനു വേണ്ടി നമ്മൾ അറെ ഉപയോഗിച്ചു. പക്ഷെ അറെ പ്രഖ്യാപനത്തോടൊപ്പം തന്നെ അറെയുടെ വലുപ്പവും നൽകിയിരിക്കണം. ഇത് മെമ്മറി അനാവശ്യമായി നഷ്ടപ്പെടുത്തുകയും ആവശ്യത്തിന് മെമ്മറി ലഭിക്കാതിരിക്കാൻ കാരണമാവുകയും ചെയ്യുന്നു. എന്നാൽ പോയിന്റർ വേരിയബിളുപയോഗിച്ച് ഡൈനാമിക് അലോക്കേഷൻ നടത്തുമ്പോൾ അറെയുടെ വലുപ്പം കൃത്യം കുട്ടികളുടെ എണ്ണത്തിനനുസരിച്ചായിരിക്കും. ഇവിടെ മെമ്മറി പാഴാക്കപ്പെടുന്നില്ല.

പക്ഷെ ഈ തരത്തിലുള്ള മെമ്മറി ഉപയോഗത്തിന് ചില പ്രശ്നങ്ങളുണ്ട്. പ്രോഗ്രാം 1.3 എപ്പോഴും `n` ന്റെ എല്ലാ വിലകൾക്കും പ്രവർത്തിക്കണം എന്നില്ല. GCC, `avg_ht` എന്ന വേരിയബിളിന്റെ ഒരു ഔട്ട്പുട്ടും ചിലപ്പോൾ നൽകില്ല. ചില അപ്രതീക്ഷിത വിലകൾ പ്രദർശിപ്പിക്കാനും സാധ്യതയുണ്ട്. ഇതിനു കാരണം `ht_ptr` ന് പ്രാരംഭവിലയായി നൽകുന്നത് ഏതെങ്കിലും ഒരു മെമ്മറി സ്ഥാനത്തിന്റെ വിലാസമാണ്. നമ്മൾ ഉപയോഗിക്കുന്ന പോയിന്ററർ അരിത്തമാറ്റിക് ഓപ്പറേഷനും നേരത്തെ പറഞ്ഞപോലെ അനുവാദമില്ലാത്ത സ്ഥാനങ്ങൾ ഉപയോഗിക്കുന്നതിന് കാരണമായേക്കാം. ഇതെല്ലാം പ്രോഗ്രാമിന്റെ അപ്രതീക്ഷിതമായ അവസാനിപ്പിക്കലിന് കാരണമാവും. പലപ്പോഴും മെമ്മറിയിൽ സൂക്ഷിച്ച ഡാറ്റയും നഷ്ടപ്പെട്ടു പോകും, അങ്ങനെ നമുക്ക് കൃത്യമായ ഔട്ട്പുട്ടും ലഭിക്കുകയില്ല. ഈ പ്രശ്നങ്ങൾ പരിഹരിക്കാൻ നമുക്ക് ഭാഗം 1.5 ലെ ഡൈനാമിക് അറെകൾ ഉപയോഗപ്പെടുത്താം.

1.4.2 പോയിന്ററിലെ റിലേഷണൽ ഓപ്പറേഷനുകൾ

നാം പഠിച്ച ആറ് റിലേഷണൽ ഓപ്പറേറ്ററുകളിൽ `==` (Equality-തുല്യം), `!=` (Non equality-തുല്യമല്ല) എന്നീ ഓപ്പറേറ്ററുകൾ മാത്രമേ പോയിന്റർ വേരിയബിളുകളിൽ ഉപയോഗിക്കാറുള്ളൂ. മെമ്മറി വിലാസം എന്നാൽ ഒരു മെമ്മറി സ്ഥാനം തിരിച്ചറിയാനുള്ള സവിശേഷ സംഖ്യയാണ്. അതിനാൽ ഇതിനു മുകളിൽ മറ്റ് റിലേഷനിൽ ഓപ്പറേറ്ററുകൾ ഉപയോഗിക്കുന്നതിൽ അർഥമില്ല. `p`, `q` എന്നിവ രണ്ട് പോയിന്ററുകൾ ആണെങ്കിൽ അവയിൽ തുല്യമായതോ/വ്യത്യസ്തമായതോ ആയി മെമ്മറി സ്ഥാനങ്ങളുടെ വിലാസങ്ങളാണ് ഉണ്ടാവുക. ഇത് പരിശോധിക്കാൻ `p==q` അല്ലെങ്കിൽ `p!=q` എന്നീ പദപ്രയോഗങ്ങൾ ഉപയോഗിക്കാം.

1.5 അറെയും പോയിന്ററും

നിങ്ങളുടെ പുരോഗതി അറിയാം

1. C++ ലെ ഡൈനാമിക് മെമ്മറി അലോക്കേഷൻ ഓപ്പറേറ്റർ ആണ് _____.
2. ചുവടെ കൊടുത്ത പ്രസ്താവന പ്രവർത്തിക്കുമ്പോൾ എന്ത് സംഭവിക്കുന്നു?
   ```
   int *b = new int(10);
   ```
3. ഓർഫൻസ് മെമ്മറി ബ്ലോക്ക് എന്നാലെന്ത്?
4. `p` ഒരു ഇന്റിജർ പോയിന്ററാണെന്നിരിക്കട്ടെ. ചുവടെ കൊടുത്തവയിൽ ഏതൊക്കെയാണ് അസാധുവായത്?

   a. `cout<<&p;` b. `p=p*5;` c. `p>0`
   d. `p++;` e. `p=1500;` f. `cout<<*p * 2;`

ഒരു പേരിൽ ഒരേ തരത്തിലുള്ള ഒന്നിലധികം ഡാറ്റകളെ ശേഖരിക്കാൻ അറെയ്ക്ക് കഴിയുമെന്ന് നാം പഠിച്ച് കഴിഞ്ഞല്ലോ. തുടർച്ചയായ മെമ്മറി സ്ഥാനങ്ങളിൽ ഡാറ്റ ശേഖരിക്കുന്നതിനു അറെ ഉപയോഗിക്കാം. ചിത്രം 1.9 കാണിക്കുന്നത് `ar[10]` എന്ന `int` ടൈപ്പ് അറെയുടെ ചിത്രമാണ്. ഇതിൽ 10 സംഖ്യകളുണ്ട്.

ഈ അറെ 1000 എന്ന മെമ്മറി ലൊക്കേഷനിൽ തുടങ്ങുന്നതായി സങ്കൽപ്പിക്കുക. ഇതിനകത്തെ ഓരോ സ്ഥാനവും 4 ബൈറ്റ് വീതം ആണ് (GCC അനുസരിച്ച്). ഈ അറെയിലെ ഏത് അംഗത്തെയും അറെയുടെ പേരും സബ്സ്ക്രിപ്റ്റും ഉപയോഗിച്ച് സൂചിപ്പിക്കാം. ഉദാഹരണം `ar[0]` എന്നത് 34 എന്ന സംഖ്യ നൽകുന്നു. `ar[1]` 12 ഉം `ar[9]` 19 ഉം തിരിച്ചുതരുന്നു.

`ptr` എന്നത് ഒരു ഇന്റിജർ പോയിന്റർ ആണെങ്കിൽ `ar` എന്ന

	ar
1000	34
1004	12
1008	8
1012	18
1016	24
1020	38
1024	43
1028	14
1032	7
1036	19

ചിത്രം 1.9: `ar` എന്ന അറെയുടെ മെമ്മറി നീക്കിവയ്ക്കൽ

നമുക്ക് ചെയ്യാം

`ar` ഈ അറെയിലെ 10 സംഖ്യകളും പ്രദർശിപ്പിക്കുന്നതിനുള്ള C++ പ്രോഗ്രാം എഴുതുക.

എങ്ങനെയാണ് ഈ അറെയുടെ ഒന്നാമത്തെ സ്ഥാനത്തിന്റെ വിലാസം ഒരു പോയിന്ററിൽ ശേഖരിക്കുന്നത്?

അറെയുടെ ആദ്യത്തെ സ്ഥാനത്തിന്റെ വിലാസം താഴെ കൊടുക്കുന്ന രീതിയിൽ പോയിന്ററിൽ ശേഖരിക്കാം.

```
ptr = &ar[0];
```

ഇനി നമുക്ക് ചുവടെ ചേർത്ത വാചകങ്ങളുടെ ഔട്ട്പുട്ട് പരിശോധിക്കാം.

```
cout<<ptr;      //1000 പ്രദർശിപ്പിക്കുന്നു, ar[0] യുടെ വിലാസം
cout<<*ptr;     //34 പ്രദർശിപ്പിക്കുന്നു, ar[0] യുടെ വില
cout<<(ptr+1);  //1004 പ്രദർശിപ്പിക്കുന്നു, ar[1] ന്റെ വിലാസം
cout<<*(ptr+1); //12 പ്രദർശിപ്പിക്കുന്നു, ar[1] ന്റെ വില
cout<<(ptr+9);  //1036 പ്രദർശിപ്പിക്കുന്നു, ar[9] ന്റെ വിലാസം
cout<<*(ptr+9); //19 പ്രദർശിപ്പിക്കുന്നു, ar[9] ന്റെ വില
```

`cout<<ar;` എന്ന പ്രസ്താവനയുടെ ഔട്ട്പുട്ട് നിങ്ങൾക്ക് കണ്ടെത്താൻ കഴിയുമോ?

ഔട്ട് പുട്ട് 1000 ആണ്. ഇത് അറെയുടെ ആദ്യത്തെ സ്ഥാനത്തിന്റെ വിലാസമാണ്. ഈ വിലാസത്തെ അറെയുടെ ബേസ് അഡ്രസ്സ് എന്ന് പറയുന്നു. വിലാസം ശേഖരിക്കുന്ന വേരിയബിൾ ഒരു പോയിന്ററാണ് എന്ന് നമുക്കറിയാം. അതുകൊണ്ട് അറെയുടെ പേരായ `ar` ഉം ഒരു പോയിന്ററാണ്. ആയതിനാൽ താഴെ കൊടുത്ത വാചകങ്ങൾ ശരിയാണ്.

```
cout<<ar;       //1000 പ്രദർശിപ്പിക്കുന്നു, ar[0] യുടെ വിലാസം
ptr=ar;         //ptr=&ar[0] എന്നതിന് തുല്യം;
cout<<*ar;      //34 പ്രദർശിപ്പിക്കുന്നു, cout<<ar[0]; എന്നതിന് തുല്യം
```

```
cout<<(ar+1); //1004 പ്രദർശിപ്പിക്കുന്നു, ar[1]; ന്റെ വിലാസം
cout<<*(ar+1);  //34 പ്രദർശിപ്പിക്കുന്നു, cout<<ar[1]; എന്നതിന് തുല്യം
```

ചുവടെ കൊടുത്ത C++ വാചകങ്ങൾ ഈ അറെയുടെ എല്ലാ അംഗങ്ങളേയും പ്രദർശിപ്പിക്കുന്നു.

```
for (int i=0; i<10; i++)
    cout<<*(ar+i)<<'\t';
```

സാധാരണ പോയിന്ററും അറെയുടെ പേരും തമ്മിൽ ചില വ്യത്യാസങ്ങളുണ്ട്. `ptr++;` എന്ന വാചകം ശരിയാണ് ഇത് `ptr=ptr+1;` എന്നതിന് തുല്യമാണ്. ഈ വാചകം പ്രവർത്തിപ്പിച്ചു കഴിഞ്ഞാൽ `ptr`, `ar[1]` ന്റെ മെമ്മറി സ്ഥാനത്തേക്ക് പോയിന്റ് ചെയ്യും. അതായത് `ptr` ൽ `ar[1]` ന്റെ വിലാസം ആണ് ശേഖരിക്കപ്പെടുന്നത്. `ptr++`ന് പകരമായി `ar++` എന്ന് ഉപയോഗിക്കാൻ കഴിയില്ല. കാരണം അറെയുടെ പേര് എപ്പോഴും അറെയുടെ ബേസ് അഡ്രസ്സ് മാത്രം ഉൾക്കൊള്ളുന്നു. ഇത് മാറ്റാൻ സാധ്യമല്ല.

ഡൈനാമിക് അറെ

C++ ൽ അറെ ഒരേ തരത്തിലുള്ള ഡാറ്റ കൈകാര്യം ചെയ്യാൻ ഉപയോഗിക്കുന്നു. പക്ഷെ ഡാറ്റയുടെ എണ്ണം മുൻകൂട്ടി അറിയില്ല എങ്കിൽ അറെ പ്രഖ്യാപിക്കാൻ പ്രയാസം നേരിടുന്നു. ഒരു ജില്ലയിലെ എല്ലാ ഹയർ സെക്കണ്ടറി സ്കൂളുകളിലെയും വിജയ ശതമാനം ശേഖരിക്കുന്നതിന് ഒരു അറെ പ്രഖ്യാപിക്കുന്നത് എങ്ങനെയാണ്? `float pass[n];` `float pass[];` എന്നീ രണ്ട് പ്രഖ്യാപനങ്ങളും C++ ൽ തെറ്റാണ്. അറെയുടെ വലുപ്പം ഒരു ഇന്റിജർ കോൺസ്റ്റന്റായിരിക്കണം. അങ്ങനെ പ്രഖ്യാപിക്കുമ്പോൾ മെമ്മറി പാഴാക്കപ്പെടുവാനോ തികയാതെ വരുവാനോ കാരണമാകാം. ഡിസ്ട്രിക്ടിലെ സ്കൂളുകളുടെ എണ്ണം പ്രോഗ്രാം തയാറാക്കുമ്പോൾ അറിയില്ല എങ്കിൽ, ഇൻപുട്ടിനനുസരിച്ച് മെമ്മറി അലോക്കേറ്റ് ചെയ്യുന്ന ഒരു അറെ ഉപയോഗിച്ചാൽ മതിയാകും. ഇവിടെയാണ് ഡൈനാമിക് അറെയുടെ ആവശ്യം ഉണ്ടാകുന്നത്. ഡൈനാമിക് അറെ പ്രോഗ്രാം പ്രവർത്തിക്കുമ്പോഴാണ് നിർമിക്കപ്പെടുന്നത്. ഇതിനുവേണ്ടി *ഡൈനാമിക് മെമ്മറി* അലോക്കേഷൻ ഓപ്പറേറ്ററായ `new` ഉപയോഗിക്കുന്നു. ഇതിന്റെ വാക്യഘടന ചുവടെ ചേർക്കുന്നു.

```
pointer = new data_type[size];
```

ഇവിടെ `size` ഒരു സ്ഥിരവില (constant), വേരിയബിൾ, ഇന്റിജർ എക്സ്പ്രഷൻ ഇവയിൽ ഏതെങ്കിലും ഒന്നായാൽ മതി.

പ്രോഗ്രാം 1.4, ഡൈനാമിക് അറെയുടെ ആശയം വിവരിക്കുന്നു. ഈ പ്രോഗ്രാമിൽ വിജയശതമാനം ശേഖരിക്കുന്നത് ഉപയോക്താവ് റൺ ടൈമിൽ നൽകുന്ന സ്കൂളിന്റെ എണ്ണമനുസരിച്ചാണ്.

Program 1.4: ഏറ്റവും കൂടിയ വിജയ ശതമാനം കാണാൻ

```
#include <iostream>
using namespace std;
int main()
{
    float *pass, max;
```

```
    int i,n;
    cout<<"Enter the number of schools: ";
    cin>>n;     //സ്കൂളിന്റെ എണ്ണം ഇൻപുട്ട് ചെയ്യാൻ
    pass = new float[n];//n അംഗങ്ങളുള്ള ഡൈനാമിക് അറെയുടെ പ്രഖ്യാപനം
    for (i=0; i<n; i++)
    {
        cout<<"Percent of pass by school "<<i+1<<": ";
        cin>>pass[i];  //സബ്സ്ക്രിപ്റ്റഡ് വേരിയബിളിന്റെ ഉപയോഗം
    }
    max=pass[0];
    for (i=1; i<n; i++)
        if (pass[i]>max) max = *(pass+i);
    /* അംഗങ്ങളെ സബ്സ്ക്രിപ്റ്റ് ഉപയോഗിച്ചും പോയിന്റർ ഓപ്പറേഷനിലൂടെയും ഉപയോഗിക്കുന്നു. */
    cout<<"Highest percent is "<<max;
    return 0;
}
```

Output:

```
Enter the number of schools: 5
Percent of pass by school 1: 75.6
Percent of pass by school 2: 66.5
Percent of pass by school 3: 89.3
Percent of pass by school 4: 71
Percent of pass by school 5: 70.6
Highest percent is 89.3
```

പ്രോഗ്രാം 1.4 ൽ ഡൈനാമിക് അറെ ഉപയോഗിച്ചാണ് ഡാറ്റ ശേഖരിക്കുന്നത്. മെമ്മറി നീക്കിവയ്ക്കൽ നടക്കുന്നത് റൺ ടൈമിൽ മാത്രമാണ്. 4 ബൈറ്റ് വീതമുള്ള 5 സ്ഥാനങ്ങൾ pass എന്ന അറെയിൽ റൺ ടൈമിൽ തയാറാക്കുന്നു. ഈ അറെയിലെ അംഗങ്ങളെ പോയിന്ററോ സബ്സ്ക്രിപ്റ്റോ ഉപയോഗിച്ച് എടുക്കാൻ കഴിയും.

നമുക്ക് ചെയ്യാം

പ്രോഗ്രാം 1.4 തയാറാക്കുന്ന സമയത്ത് ജില്ലയിലെ സ്കൂളുകളുടെ എണ്ണം കൃത്യമായി പ്രോഗ്രാമർക്ക് അറിയില്ല. ഇവിടെ സാധാരണ അറെ ഉപയോഗിക്കുകയാണെങ്കിൽ ഒരു കൂടിയ സംഖ്യ വലുപ്പമായി നൽകി അറെ പ്രഖ്യാപിക്കേണ്ടതായി വരുകയും ഇത് ശേഷിക്കുന്ന മെമ്മറി പാഴാക്കാൻ കാരണമാവുകയും ചെയ്യും. എന്നാൽ ഇവിടെ ഡൈനാമിക് അറെ ഉപയോഗിച്ചാൽ ഉപയോക്താവ് നൽകിയ കൃത്യമായ എണ്ണത്തിൽ അറെ നിർമിക്കുന്നതിനാൽ മെമ്മറി പാഴാക്കപ്പെടുന്നില്ല.

നമുക്ക് ചെയ്യാം

താഴെയുള്ള രണ്ട് പ്രസ്താവനകൾ തമ്മിലുള്ള വ്യത്യാസം എഴുതുക:

```
int *ptr = new int(10);
int *ptr = new int[10];
```

നിങ്ങളുടെ പുരോഗതി അറിയാം

1. ഡൈനാമിക് അറെ എന്നാൽ എന്ത്?
2. ഒരു അറെയിലെ ഒന്നാമത്തെ ലൊക്കേഷന്റെ വിലാസത്തെ _____ എന്ന് പറയുന്നു.
3. `arr` ഒരു ഇന്റിജർ അറെയാണെങ്കിൽ താഴെ കൊടുത്തവയിൽ തെറ്റേത്?

   a. `cout<<arr;` b. `arr++;` c. `cout<<*(arr+1);`

   d. `cin>>arr;` e. `arr=1500;` f. `cout<<*arr * 2;`
4. 10 പുസ്തകങ്ങളുടെ പേരുകൾ പോയിന്ററർ ഉപയോഗിച്ച് സൂചിപ്പിക്കാനുള്ള C++ പ്രഖ്യാപന പ്രസ്താവന എഴുതുക.
5. ഒരു പോയിന്റർ പ്രഖ്യാപിച്ച് അതിൽ നിങ്ങളുടെ പേര് പ്രാരംഭ വിലയായി ചേർക്കുക.

1.7 പോയിന്ററും സ്ട്രക്ചറും

ഈ അധ്യായത്തിന്റെ തുടക്കത്തിൽ നമ്മൾ സ്ട്രക്ചർ ഡാറ്റാതരത്തെക്കുറിച്ചും അതിന്റെ ഉപയോഗത്തെക്കുറിച്ചും ചർച്ച ചെയ്തു. ഈ ഭാഗത്ത് നമ്മൾ സ്ട്രക്ചർ വേരിയബിളിനെ പോയിന്ററിന്റെ സഹായത്താൽ എങ്ങനെ ഉപയോഗിക്കാമെന്ന് പഠിക്കുന്നു. ഒരു employee സ്ട്രക്ചർ ചുവടെ ചേർക്കുന്നു.

```
struct employee
{
    int ecode;
    char ename[15];
    float salary;
};
```

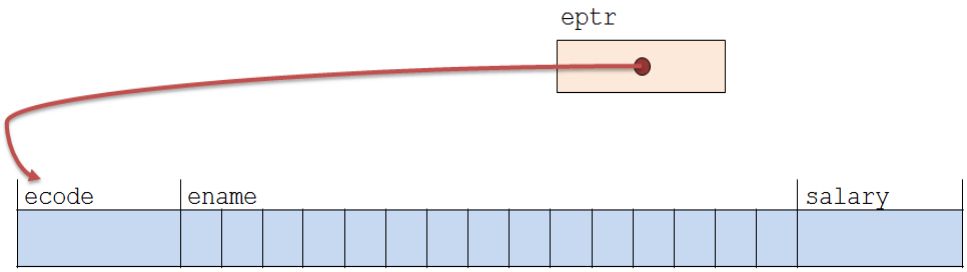
ഇനി ചുവടെ കൊടുത്തിരിക്കുന്ന പ്രഖ്യാപന പ്രസ്താവന പരിശോധിക്കാം.

```
employee *eptr;
```

ഇവിടെ `eptr` എന്ന പോയിന്ററിന് ഒരു `employee` സ്ട്രക്ചറിന്റെ അഡ്രസ്സ് ഉൾക്കൊള്ളാൻ കഴിയും. ഇനി താഴെ കൊടുത്ത പ്രസ്താവന ശ്രദ്ധിക്കുക.

```
eptr = new employee;
```

ഈ വാചകം 23 ബൈറ്റ് മെമ്മറി നീക്കിവയ്ക്കുകയും ചെയ്യുകയും അതിന്റെ ബേസ് അഡ്രസ്സ് `eptr` എന്ന പോയിന്ററിൽ ശേഖരിക്കുകയും ചെയ്യും. ചിത്രം 1.11 ൽ ഇക്കാര്യം വിവരിക്കുന്നു.



ചിത്രം 1.11: employee ടൈപ്പ് ഡാറ്റയുടെ ഡൈനാമിക് അലോക്കേഷൻ

സ്ട്രക്ചർ അംഗങ്ങളെ താഴെ കൊടുത്ത വിധമാണ് ഉപയോഗിക്കുന്നത് എന്ന് നമുക്കറിയാം.

```
structure_variable.element_name
```

ecode, ename, salary എന്നീ അംഗങ്ങളെ ഉപയോഗിക്കാൻ നമുക്ക് ഇവിടെ സ്ട്രക്ചർ വേരിയബിൾ ഇല്ല. eptr എന്ന പോയിന്ററിലൂടെയാണ് ഇവയെ ഉപയോഗിക്കേണ്ടത്. ഇതിന്റെ വാക്യഘടന ചുവടെ ചേർക്കുന്നു.

```
structure_pointer->element_name
```

ശ്രദ്ധിക്കുക, സ്ട്രക്ചർ പോയിന്ററും അംഗങ്ങളും '->' ഓപ്പറേറ്ററിലൂടെ '->' ബന്ധിപ്പിക്കപ്പെട്ടിരിക്കുന്നു. ഈ ഓപ്പറേറ്റർ '-' നു പുറകെ '>' ചിഹ്നം കൂടി ഉപയോഗിച്ചാണ് എഴുതുന്നത്. ചുവടെ കൊടുത്ത പ്രസ്താവനകൾ ചിത്രം 1.11 ലെ സ്ട്രക്ചർ അംഗങ്ങളെ ആക്സസ് ചെയ്യാൻ ഉപയോഗിക്കാം.

```
eptr->ecode = 657346;   //employee code ന് വില നൽകുന്നു
gets(eptr->ename);      //employee യുടെ പേര് ഇൻപുട്ട് ചെയ്യുന്നു
cin>> eptr->salaray;    //employee യും സാലറി ഇൻപുട്ട് ചെയ്യുന്നു
cout<< eptr->salary * 0.12; //സാലറിയുടെ 12% പ്രദർശിപ്പിക്കുന്നു
```

നമുക്ക് ചെയ്യാം

1.3.1 എന്ന ഭാഗത്തിൽ നമ്മൾ cx_ptr എന്ന പോയിന്റർ വേരിയബിൾ complex ടൈപ്പ് സ്ട്രക്ചറിൽ ഉപയോഗിച്ചു. ഒരു complex നമ്പർ ഇൻപുട്ട് ചെയ്ത് അതിനെ യഥാർഥ രൂപത്തിൽ പ്രദർശിപ്പിക്കാനുള്ള C++ പ്രസ്താവനകൾ സ്ട്രക്ചർ പോയിന്റർ ഉപയോഗിച്ച് എഴുതുക.

നമുക്ക് ഒരു പുതിയ അംഗത്തെ കൂട്ടിച്ചേർത്തുകൊണ്ട് employee സ്ട്രക്ചറിനെ പരിഷ്കരിക്കാം.

```
struct employee
{
    int ecode;
    char ename[15];
    float salary;
    int *ip;
};
```

തീർച്ചയായും ip എന്ന അംഗം ഒരു ഇന്റിജർ പോയിന്ററാണ്. ഇതിന് ഒരു ഇന്റിജർ സ്ഥാനത്തിന്റെ വിലാസം സൂക്ഷിക്കാം. ചുവടെ കൊടുത്ത പ്രസ്താവനകൾ പോയിന്റർ ip യുടെ ഉപയോഗം വിവരിക്കുന്നു.

```
eptr->ip = new int(5); /* ഡൈനാമിക് അലോക്കേഷൻ നടത്തി ip യെ ഒരു ഇന്റിജർ ലൊക്കേഷനുമായി ബന്ധിപ്പിച്ച് അതിൽ 5 ശേഖരിക്കുന്നു */ .
cout << *(eptr->ip);     // 5 എന്ന വില പ്രദർശിപ്പിക്കുന്നു
int n = eptr->*ip+1;     // 5 നോട് 1 കൂട്ടി n ൽ ശേഖരിക്കുന്നു
```

നോക്കുക ip പോയിന്റ് ചെയ്യുന്ന വില രണ്ട് രീതിയിൽ പുറത്തെടുക്കാം. *(eptr->ip) അല്ലെങ്കിൽ eptr->*ip. ഒരു സ്ട്രക്ച്ചറിന് ഏത് തരത്തിലുള്ള പോയിന്റിനേയും അംഗങ്ങളാക്കാം. വേണമെങ്കിൽ ഒരു സ്ട്രക്ചറിന്റെ തരത്തിൽ തന്നെയുള്ള പോയിന്ററിനെ അതിന് അംഗമാക്കാം.

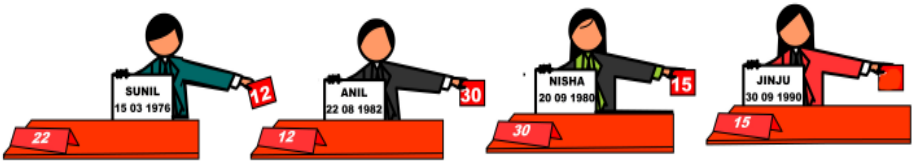
```
struct employee
{
    int ecode;
    char ename[15];
    float salary;
    employee *ep;
};
```

ep എന്ന അംഗം employee തരത്തിലുള്ള പോയിന്റർ ആണ്.

ഇപ്പോൾ സ്ട്രക്ചർ employee യെ സ്വയം സൂചിത സ്ട്രക്ചർ (Self referential structure) എന്ന് വിളിക്കുന്നു. ഇതിനെക്കുറിച്ച് നമുക്ക് വിശദമായി ചർച്ച ചെയ്യാം.

സ്വയം സൂചിത സ്ട്രക്ചർ (Self referential structure)

സ്വയം സൂചിത സ്ട്രക്ചർ ഒരു പ്രത്യേക തരം സ്ട്രക്ച്ചറാണ്. ഇതിൽ ഇതേ സ്ട്രക്ചർ തരത്തിലുള്ള ഒരു പോയിന്റർ വേരിയബിൾ ഉണ്ടായിരിക്കും. ഈ പോയിന്റിന് ഇതേ പോലുള്ള മറ്റൊരു സ്ട്രക്ചറിലേക്ക് പോയിന്റ് ചെയ്യാം. ഇങ്ങനെ നിർമിക്കുന്ന കണ്ണി എത്ര വേണമെങ്കിലും ദീർഘിപ്പിക്കാം. ചിത്രം 1.12 ഈ ആശയം വിവരിക്കുന്നു.



ചിത്രം 1.12: ഒരു employee സ്ട്രക്ചർ മറ്റൊന്നിനെ പോയിന്റ് ചെയ്യുന്നു

"Sunil" എന്ന എംപ്ലോയി "Anil" എന്ന എംപ്ലോയിയെ പോയിന്റ് ചെയ്യുന്നു. അയാളുടെ ടേബിൾ നമ്പർ 12 ആണ്. "Anil" പോയിന്റ് ചെയ്യുന്നത് "Nisha" എന്ന എംപ്ലോയിയെയാണ് ഇത് തുടരുന്നു.

C, C++ ഭാഷകളിൽ ലിങ്ക്ഡ്‌ലിസ്റ്റ്, ട്രീ തുടങ്ങിയ ഡയനാമിക് ഡാറ്റാ സ്ട്രക്ചറുകളുടെ നിർമാണത്തിന് സഹായകമായ ഒരു പ്രധാന ടൂൾ ആണ് സ്വയം സൂചിത സ്ട്രക്ചർ. പ്രോഗ്രാമിന്റെ പ്രവർത്തന സമയത്ത് നീക്കി വയ്ക്കുന്ന മെമ്മറി സ്ഥാനങ്ങളിലെ ഡാറ്റയുടെ കൂട്ടത്തെ ഡൈനാമിക് ഡാറ്റാ സ്ട്രക്ചർ എന്നു വിളിക്കുന്നു. ലിങ്ക്ഡ് ലിസ്റ്റ് ഡാറ്റാ സ്ട്രക്ചറിനെക്കുറിച്ച് കൂടുതലായി അധ്യായം 3 ൽ ചർച്ച ചെയ്യുന്നു.

നമുക്ക് സംഗ്രഹിക്കാം

C++ ലെ നൂതനമായ ഡാറ്റാതരങ്ങളെ കുറിച്ച് നാം ചർച്ച് ചെയ്തു. ഗ്രൂപ്പ് ഡാറ്റയെ ഒരു പേരിൽ പ്രതിനിധാനം ചെയ്യാൻ സ്ട്രക്ചർ ഡാറ്റാതരം ഉപയോഗിച്ചു. സ്ട്രക്ചർ അംഗങ്ങളെ ഡോട്ട് ഓപ്പറേറ്ററി (.) ലൂടെ ഉപയോഗിക്കുന്നത് നാം ചർച്ച ചെയ്തു. പോയിന്റർ പ്രത്യേക ഡാറ്റാ തരം ആണെന്ന് മനസ്സിലാക്കി. പോയിന്ററുമായി ബന്ധപ്പെട്ട ഓപ്പറേഷനുകൾ പദപ്രയോഗത്തിന്റെ സഹായത്തോടെ വിവരിച്ചു. ഡൈനാമിക് മെമ്മറി അലോക്കേഷൻ എന്ന ആശയവും, ആവശ്യമായ ഓപ്പറേറ്ററുകളും നേട്ടങ്ങളും ചർച്ച ചെയ്തു. അറെയും പോയിന്ററും തമ്മിലുള്ള ബന്ധം വിവരിച്ചു. സ്ട്രിങ് ഡാറ്റ പോയിന്ററിലൂടെ ഉപയോഗിക്കുന്നതും പോയിന്ററിനേയും സ്ട്രക്ചറിനേയും ബന്ധിപ്പിക്കുന്ന വിധവും മനസ്സിലാക്കി. ഈ അധ്യായത്തിൽ പഠിച്ച പല കാര്യങ്ങളും അധ്യായം - 3 പഠിക്കുന്നതിനുള്ള അടിസ്ഥാനമായി ഉപയോഗിക്കാം.

നമുക്ക് പരിശീലിക്കാം

1. ടെലിഫോൺ വരിക്കാരന്റെ വിവരങ്ങൾ അടങ്ങുന്ന ഒരു സ്ട്രക്ചർ നിർവചിക്കുക. അതിൽ പേര്, ടെലിഫോൺ നമ്പർ എന്നീ അംഗങ്ങൾ ഉണ്ടായിരിക്കണം. ഇതുപയോഗിച്ച് പേര് നൽകിയാൽ നമ്പർ തരുന്നതും, അല്ലെങ്കിൽ നമ്പർ നൽകിയാൽ പേര് തരുന്നതുമായ ഒരു മെനു നിയന്ത്രിതമായ C++ പ്രോഗ്രാം തയാറാക്കുക.
2. ഒരു ബാങ്കിലെ ഉപഭോക്താവിന്റെ വിവരങ്ങൾ അടങ്ങുന്ന സ്ട്രക്ചർ നിർവചിക്കുക. അക്കൗണ്ട് നമ്പർ, പേര്, അക്കൗണ്ട് ഓപ്പണിംഗ് തീയതി, ബാലൻസ് എന്നീ വിവരങ്ങൾ ഉണ്ടായിരിക്കണം. ഇതുപയോഗിച്ച് ഡെപ്പോസിറ്റ്, വിഡ്രോ, വ്യൂ എന്നീ പ്രവൃത്തികൾ നടത്താനാവശ്യമായ ഒരു മെനു നിയന്ത്രിത പ്രോഗ്രാം തയാറാക്കുക. ഡെപ്പോസിറ്റും വിഡ്രോവലും നടക്കുമ്പോൾ ബാലൻസ് സംഖ്യയിൽ മാറ്റം വരണം. അക്കൗണ്ടിൽ മിനിമം ബാലൻസായി Rs. 1000/- ഉണ്ടായിരിക്കണം (അക്കൗണ്ട് ഓപ്പണിംഗ് തീയതി നെസ്റ്റെഡ് സ്ട്രക്ചറായി ഉൾപ്പെടുത്താം).
3. വിദ്യാർഥികൾക്ക് കമ്പ്യൂട്ടർ സയൻസിൽ കിട്ടിയ TE സ്കോർ അവരോഹണ ക്രമത്തിൽ ക്രമീകരിക്കുന്നതിനു പോയിന്റർ ഉപയോഗിച്ച് ഒരു C++ പ്രോഗ്രാം എഴുതുക.
4. ക്യാരക്ടർ പോയിന്റർ ഉപയോഗിച്ച് ഇൻപുട്ട് ചെയ്ത ഒരു സ്ട്രിങ് പാലിൻഡ്രോം ആണോ അല്ലയോ എന്ന് പരിശോധിക്കാനുള്ള ഒരു C++ പ്രോഗ്രാം തയാറാക്കുക.
5. പോയിന്റർ ഉപയോഗിച്ച്, നിങ്ങളുടെ ക്ലാസ്സിലെ എല്ലാ കുട്ടികളുടെയും പേരുകൾ ഇൻപുട്ട് ചെയ്യുക. ഇതിൽ നിന്നും ഒരു റോൾ ലിസ്റ്റ് നിർമിക്കുക. ഈ ലിസ്റ്റിൽ പേരുകൾ അക്ഷരമാല ക്രമത്തിലും റോൾ നമ്പർ 1, 2, 3 ... എന്ന ക്രമത്തിലും വരണം.
6. രജിസ്റ്റർ നമ്പർ, പേര്, ആറ് വിഷയത്തിലെ CE മാർക്കുകൾ എന്നീ വിവരങ്ങൾ അടങ്ങുന്ന ഒരു student സ്ട്രക്ചർ നിർവചിക്കുക. സ്ട്രക്ചർ പോയിന്റർ ഉപയോഗിച്ച് ഈ വിവരങ്ങൾ ഇൻപുട്ട് ചെയ്യാനും രജിസ്റ്റർ നമ്പർ, പേര്, സി.ഇ. മാർക്കുകളുടെ തുക എന്നിവ പ്രദർശിപ്പിക്കാനും വേണ്ട പ്രോഗ്രാം എഴുതുക.

നമുക്ക് വിലയിരുത്താം

1. C++ ലെ അറെയും സ്ട്രക്ചറും താരതമ്യം ചെയ്യുക.
2. താഴെ കൊടുത്തിരിക്കുന്ന സ്ട്രക്ചർ നിർവചനത്തിലെ തെറ്റുകൾ കണ്ടെത്തി അതിനുള്ള കാരണമെഴുതുക.

```
struct
{
    int roll, age;
    float fee=1000;
};
```

3. താഴെ കൊടുത്തിരിക്കുന്ന സ്ട്രക്ചർ നിർവചനം വായിച്ച് ചോദ്യങ്ങൾക്ക് ഉത്തരം എഴുതുക.

```
struct book
{
    int book_no;
    char bk_name[20];
    struct
    {
        short dd;
        short mm;
        short yy;
    }dt_of_purchase;
    float price;
};
```

   a. ഈ സ്ട്രക്ചറിലെ വിവരങ്ങൾ സൂചിപ്പിക്കാൻ book ടൈപ്പിൽ ഒരു വേരിയബിൾ പ്രഖ്യാപിക്കാനുള്ള C++ പ്രസ്താവന എഴുതുക. വേരിയബിളിന് എത്ര മെമ്മറി ആവശ്യമുണ്ട്. ന്യായീകരിക്കുക.

   b. നിങ്ങളുടെ കമ്പ്യൂട്ടർ സയൻസ് ടെക്സ്റ്റ് ബുക്കിന്റെ വിവരങ്ങൾ ഉപയോഗിച്ച് ഈ സ്ട്രക്ചറിന് പ്രാരംഭവില നൽകാനുള്ള C++ പ്രസ്താവന എഴുതുക.

   c. book ന്റെ വിവരങ്ങൾ പ്രദർശിപ്പിക്കാൻ ആവശ്യമായ C++ പ്രസ്താവനകൾ എഴുതുക.

   d. "അകത്തുള്ള സ്ട്രക്ചർ ടാഗിന്റെ അഭാവം ഒരു തെറ്റും വരുത്തുന്നില്ല" - ഈ വാചകം ശരിയോ തെറ്റോ എന്ന് കാരണസഹിതം സമർഥിക്കുക.

4. "സ്ട്രക്ചർ ഒരു ഉപഭോക്ത്യനിർവചിത ഡാറ്റാതരമാണ്" - ഉദാഹരണ സഹിതം സമർഥിക്കുക.

5. താഴെ കൊടുത്തിരിക്കുന്ന വാചകങ്ങൾ വായിക്കുക:

   i. C++ ൽ സ്ട്രക്ചർ നിർവചിക്കുമ്പോൾ ടാഗ് ഒഴിവാക്കാം.

   ii. ഒരു സ്ട്രക്ചർ വേരിയബിളിലെ ഡാറ്റ മറ്റൊരു സ്ട്രക്ചർ വേരിയബിളിലേക്ക് കോപ്പി ചെയ്യണമെങ്കിൽ രണ്ട് സ്ട്രക്ച്ചർ വേരിയബിളുകളും ഒരേ ടാഗിൽ നിർവചിച്ചിരിക്കണം.

   iii. സ്ട്രക്ചർ അംഗങ്ങളെ structure_name.element എന്ന രീതിയിലാണ് സൂചിപ്പിക്കുന്നത്.

   iv. ഒരു സ്ട്രക്ചറിന് മറ്റൊരു സ്ട്രക്ചറിനെ ഉൾക്കൊള്ളാം.

   ഇനി താഴെ കൊടുത്തവയിൽ ശരിയായത് തിരഞ്ഞെടുക്കുക.

   a. (i) ഉം (ii) ഉം വാചകങ്ങൾ ശരിയാണ്

   b. (ii) ഉം (iv) ഉം വാചകങ്ങൾ ശരിയാണ്

   c. (i) ഉം, (ii) ഉം, (iv) ഉം വാചകങ്ങൾ ശരിയാണ്

   d. എല്ലാ വാചകങ്ങളും ശരിയാണ്

6. ചുവടെ ചേർത്തിരിക്കുന്ന C++ പ്രസ്താവനകൾ വായിക്കുക.

```
int * p, a=5;
p=&a;
```

   a. p എന്ന വേരിയബിളിന്റെ പ്രത്യേകതയെന്ത്?

   b. രണ്ടാമത്തെ വാചകം പ്രവർത്തിപ്പിച്ചതിനുശേഷം p യുടെ ഉള്ളടക്കം എന്തായിരിക്കും.

   c. *p+1, *(p+1) എന്നീ പദപ്രയോഗങ്ങൾ എങ്ങനെ വ്യത്യാസപ്പെട്ടിരിക്കുന്നു?

7. ചുവടെ കൊടുത്തിരിക്കുന്ന C++ പ്രോഗ്രാം ശകലത്തിലെ തെറ്റ് കണ്ടുപിടിച്ച് കാരണമെഴുതുക.

```
int *p,*q, a=5;
float b=2;
p=&a;
q=&b;
cout<<p<<*p<<*a;
if (p<q)cout<<p;
   cout<<*p * a;
```

8. ഒരു പ്രോഗ്രാം തയാറാക്കുമ്പോൾ ഡൈനാമിക് അലോക്കേഷൻ രീതി ഉപയോഗിച്ചു. എന്നാൽ delete ഓപ്പറേറ്റർ ഉപയോഗിച്ചുള്ള വാചകം ആ പ്രോഗ്രാമിൽ ഇല്ലായിരുന്നു. ഇത് സൃഷ്ടിക്കുന്ന പ്രശ്നം വിവരിക്കുക.

9. താഴെ കൊടുത്തിരിക്കുന്ന C++ പ്രസ്താവനകൾ വായിച്ച് ചോദ്യങ്ങൾക്ക് ഉത്തര മെഴുതുക.

```
int mark[] = {34, 12, 25, 56, 38};
int *p = mark;
```

a. p യുടെ വിലയെന്ത്

b. *p + *(ar+2) എന്ന പദപ്രയോഗത്തിന്റെ വില എന്താണ്?

c. ar++; എന്ന വാചകം തെറ്റാണ് - എന്തുകൊണ്ട്? ഇത് p++; എന്ന വാചക ത്തിൽ നിന്നും എങ്ങനെ വ്യത്യാസപ്പെട്ടിരിക്കുന്നു.

10. താഴെ കൊടുത്തിരിക്കുന്ന പ്രോഗ്രാം ശകലത്തിന്റെ പ്രവർത്തനം വിവരിച്ച് ഔട്ട്പുട്ട് കണ്ടെത്തുക.

```
char *str = "Tobacco Kills";
for (int i=0; str[i]!='\0'; i++)
    if (i>8)
        *(str+i) = toupper(*(str+i));
cout<<str;
```

11. താഴെ കൊടുത്ത C++ പ്രസ്താവനകൾ ശ്രദ്ധിക്കുക.

```
int ar[] = {14, 29, 32, 63, 30};
```

ചുവടെ ചേർത്തവയിൽ ഒരു വാചകം 32 എന്ന അംഗത്തെ നൽകുന്നില്ല. ഏതാ ണത്?

a. ar[2] b. ar[*ar%3] c. *ar+2 d. *(ar+2)

12. new, delete എന്നീ ഓപ്പറേറ്ററുകളുടെ പ്രവർത്തനങ്ങൾ ഉദാഹരണ സഹിതം വിവരിക്കുക.

13. മെമ്മറി ലീക്ക് എന്നാലെന്ത്? എന്താണ് ഇതിന് കാരണം? ഈ സാഹചര്യം എങ്ങനെ ഒഴിവാക്കാം?

14. ചുവടെ കൊടുത്തിരിക്കുന്ന പ്രസ്താവനകൾ താരതമ്യം ചെയ്യുക.

```
int a=5;
int *a=new int(5);
```

15. ചുവടെ കൊടുത്തിരിക്കുന്ന സ്ട്രക്ചർ നിർവചനം വായിച്ച് ചോദ്യങ്ങൾക്ക് ഉത്തര മെഴുതുക.

```
struct sample
{
    int num;
    char *str;
}   *sptr;
```

a. `sample` ഡാറ്റാ തരത്തിലെ ഒരു ലൊക്കേഷനിൽ ഡൈനാമിക് അലോക്കേഷൻ നടത്തി അഡ്രസ്സ് `sptr` ൽ ശേഖരിക്കാൻ വേണ്ട C++ പ്രസ്താവന എഴുതുക.

b. `sptr` പോയിന്റ് ചെയ്യുന്ന മെമ്മറി സ്ഥാനത്തേക്ക് ഡാറ്റ ഇൻപുട്ട് ചെയ്യാനുള്ള C++ പ്രസ്താവനകൾ എഴുതുക.

c. ഈ സ്ട്രക്ചറിനെ സ്വയം സൂചിത സ്ട്രക്ചറായി നവീകരിക്കുക.