Powtórka z C cz. II

Wykład 3

Funkcje

- Dają możliwość definiowania własnych „instrukcji”
- Pozwalają na wielokrotne wykorzystanie tego samego kawałka kodu
- Umożliwiają czytelniejszą organizację kodu programu
- Funkcje wywołuje się przez podanie jej nazwy wraz z umieszczonymi w nawiasach argumentami
- Funkcje mogą lecz nie muszą zwracać wartość
  - Zawsze jednak tylko jedną
- Typ wartości zwracanej nie jest brany pod uwagę przy dopasowywaniu wersji funkcji do wywołania
  - Dlaczego?

Parametry

- Parametry występujące w deklaracji i definicji funkcji nazywamy formalnymi
  - Ich nazwy nie mają większego znaczenia
- Parametry użyte do wywołania funkcji nazywamy aktualnymi lub parametrami wywołania
- Sposób przesyłania parametrów
  - Przez wartość
    - Funkcja otrzymuje kopie obiektów (operacje na zmiennych lokalnych)
    - Nie powinno się przesyłać dużych obiektów
  - Przez referencje
    - Funkcja otrzymuje oryginalny obiekt

Deklaracja, a definicja

```
void licz()
{
    cout << 2*2;
}

int main()
{
    licz(void);
}
```

```
void licz();

int main()
{
    licz();
}

void licz()
{
    cout << 2*2;
}
```

Przykłady deklaracji

- W wywoływaniu funkcji istotny jest typ argumentów
  - Ścisła kontrola typów

```
viod swap_int(int a, int b);
viod swap_float(float a, float b);
int kwadrat(int x);
float licz(); // == float licz(void);
double kwadrat(double &, double &);
```

Zwracanie rezultatu

- W przypadku **`void`** funkcja nie zwraca żadnego rezultatu
- W każdym innym należ użyć słowa kluczowego **`return`**
  - Funkcja powinna zwracać odpowiedni typ, taki jaki został zadeklarowany
  - Jeśli typ jest niezgodny to następuje próba wykonania niejawnej standardowej konwersji
- Przykład cpp_3.1

Przesyłanie argumentów przez wartość

- Funkcja pracuje na kopiach przekazanych argumentów
  - NIE SĄ ZMIENIANE ORYGINALNE WARTOŚCI
  - Tworzenie nowych obiektów powiększa zajmowaną pamięć
  - Wywołania przez wartość nie powinno się stosować przy dużych obiektach
- Przykład cpp_3.2

Przesyłanie argumentów przez referencje

- Funkcja pracuje na oryginalnych obiektach
  - NIE SĄ TWORZONE KOPIE
  - Należy zadbać aby obiekty nie zostały przez przypadek zmieniony
  - Uwaga przy obiektach typu `const`
- Przykład cpp_3.3

Argumenty domniemane

- Stosuje się kiedy jakaś wartość argumentu jest często używana
  - `void temperatura(float T, int skala = 0)`
  - `temperatura(100);`
  - `temperatura(100, 1);`
- Domniemanych wartości może być więcej, wtedy muszą zostać umieszczone na końcu listy argumentów
  - `void licz(int x, int y = 0, int z = 0)`
  - `licz(0);`
  - `licz(1, 1);`
  - `licz(1, ,1); // BŁĄD`
- Przykład cpp_3.4

Funkcje `inline`

- Stosuje się do funkcji o małej liczbie instrukcji w celu przyspieszenia jej działania
  - W szczególności przy wielokrotnym jej wywoływaniu
- Nie występują w klasycznym C
- ```
  inline int zao(float liczba)
  {
     return (liczba + 0.5);
  }
  ```
- Funkcje typu `inline` musza być od razu zdefiniowane!!!
  ```

Rekurencja

- Z rekurencją spotykamy się kiedy funkcja wywołuje samą siebie
- ```
  long silnia (int n)
  {
      if (n<2) return 1;
      else return n*silnia(n-1);
  }
  ```
- Wywołania rekurencyjne umożliwią często prostsze zakodowanie algorytmu, ale okupione jest to wzrostem czasu jego wykonania
- Przykład cpp_3.5
  ```

Przeładowanie nazw funkcji

- W danym zakresie ważności (np. pliku) występuje więcej niż jedna funkcja o takiej samej nazwie
- Wybór funkcji odbywa się na podstawie typów i liczby argumentów z którymi funkcja jest wywoływana
  - Należy pamiętać o możliwości wystąpienia niejawnych konwersji
- Typ zwracany przez funkcję nie jest brany pod uwagę

Stosowanie przeładowania

- Przeładowanie nazw funkcji powinno stosować się w sytuacji gdy funkcja wykonuje podobne zadanie na innych typach danych
  - Zwracanie wartości min i max
  - Obliczenie średniej
  - Wypisywanie danych
  - ...
- Przykład cpp_3.6

Przeładowanie, a argumenty domniemane

- ```
  void fun(float);
  void fun(char);
  void fun(int, float = 0);
  ```

- ```
  fun(1.1);    //fun(float)
  fun('A');    //fun(char)
  fun(2);      //fun(int, float = 0)
  fun(2,2.2); //fun(int, float)
  ```

- Ale nie może istnieć w tym przypadku
  ```
  void fun(int);
  ```

Przeładowanie, a zakres ważności

- Przeładowanie następuje tylko dla tego samego zakresu ważności
- Dla różnych zakresów ważności następuję zasłanianie nazw
- W ramach obszaru lokalnego funkcje również mogą być przeładowywane
- Przykład cpp_3.7

Tablice

- Tablica jest to ciąg obiektów tego samego typu, które zajmują ciągły obszar w pamięci
- Tablice są typem pochodnym
- Rozmiar tablic musi być znany w momencie kompilacji programu
- `int tab[10];`
- `float tab[100];`
- `...`
- Elementy w tablicy są numerowane od 0 (zera)
  - Należ pamiętać, ponieważ taki fragment kodu

    ```
    int t[20]; t[20] = 0;
    ```

    jest błędny

Inicjalizowanie tablic

- Przed użyciem elementów tablicy, każdy z nich powinien zostać zainicjalizowany
  - Częsty błąd, który dość trudno jest wychwycić
- ```
  int t1[4] = {2, 4, 6, 8};  /*
  wszystkie elementy zainicjalizowane */

  int t2[4] = {2, 4}; /* ostanie dwa
  zainicjalizowane zerami */

  int t3[] = {2, 4, 6, 8};
  ```
- Przykład cpp_3.8
  ```

Przekazywanie tablicy do funkcji

- Tablicy nie da się przesłać przez wartość
- Tablice przesyła się podając funkcji tylko adres jej początku
- Nazwa tablicy jest jednocześnie adresem jej początku (zerowego elementu)
- Funkcja musi w inny sposób poznać wielkość tablicy, aby móc poprawnie wykonywać na niej operacje
- Przykład cpp_3.9

Tablice znaków

- `char tekst[] = {„Tablica znakow”};`
- Koniec tablicy znaków określony jest przez element o wartości 0 czyli `NULL`
  - Czyli tablica o np. trzech znakach będzie zawierać cztery elementy
  - Uwaga definicja `char tekst [] = {'a', 'b', 'c'};` tworzy tablice znaków z trzema elementami, a nie łańcuch
- Operatory `+`, `=`, `==` itp. nie są zdefiniowane dla całych tablic znakowych
  - Należy napisać własne funkcje do obsługi łańcuchów lub skorzystać z funkcji bibliotecznych takich jak:
  - `strcpy`, `strcmp`, `strcat` itp. (cstdio dawniej stdio.h)
- Najlepiej wykorzystać klasę `std::string` z STL (zostanie omówione później)

Tablice wielowymiarowe

- Tablice wielowymiarowe mogą być tworzone z różnych, ale określonych typów obiektów (`int`, `float` ...)
  - Np. `int t[5][2];`
  - Tworzona jest tablica typu `int` składająca się z pięciu wierszy i dwóch kolumn, a tak naprawdę tablica `int t[5]`, składająca się z tablic dwuelementowych typu `int`
  - Elementy w pamięci rozmieszczone są w ten sposób, iż najszybciej zmienna się najbardziej skrajny prawy wskaźnik
- Co oznacza `int t [i,j];`?

Przekazywanie tablic wielkowymiarowych do funkcji

- Podobnie jak poprzednio potrzebny jest:
  - Typ elementów tablicy
  - Informacja jak w pamięci rozmieszczone są poszczególne elementy
    - Czyli wartości poszczególnych wymiarów
- Przykład
  - ```
    void fun(float t[1000][2]); //(float[][2])
    float dane[1000][2];
    fun(dane);
    ```
- Możliwe jest również tworzenie tablic o większej liczbie wymiarów
  - W każdym przypadku zasada jest taka sama
    ```

Wskaźniki

- Wskaźnik służy do pokazywania na różne obiekty (różnych typów)
- Definiowanie wskaźników
  - `int *w;`
  - `float *f;`
- Wskaźnik może pokazywać na obiekt jednego konkretnego typu
- Wskaźnik tak naprawdę pokazuje na obszar pamięci, w którym dany obiekt jest przechowywany
- Wskaźnik posiadając informację o typie może zinterpretować dany obszar pamięci, w którym obiekt jest przechowywany

Używanie wskaźników

- W języku C++ definicje obiektów zapisane są tak, iż przypominają sposób wykorzystania ich w wyrażeniach
  - Np. `int *w;` pozwala na używanie wartości pokazywanej przez wskaźnik w sposób następujący: `int i = *w;`
  - Podobnie jest z omówionymi wcześniej tablicami
- Ustawienie wskaźnika wykonuje się za pomocą jednoargumentowego operatora `&` zwracającego adres obiektu
  - `int x; int *y = &x;`
- Przykład cpp_3.11

Wskaźniki z atrybutem const

- Jeżeli mamy jakiś typ o nazwie **nazwa_typu** to (czytamy od prawej do lewej)
  - **nazwa_typu*** - jest wskaźnikiem do **nazwa_typu**
  - **const nazwa_typu*** - jest wskaźnikiem do **nazwa_typu**, ale obiekt nie może być zmieniony przez ten wskaźnik
  - **nazwa_typu* const** – jest stałym wskaźnikiem do **nazwa_typu**, tzn. obiekt na który ten wskaźnik pokazuje można zmieniać, a samego wskaźnika nie
  - **const nazwa_typu* const** – jest stałym wskaźnikiem do stałego obiektu typu **nazwa_typu**, tzn. obiekt na który ten wskaźnik pokazuje nie można zmieniać i samego wskaźnika też nie
- Z referencjami jest podobnie, z tą różnicą że samych referencji nie możemy zmieniać (pierwsze dwa przykłady)

Wskaźnik typu `void`

- Normalnie wskaźnik przechowuje informacje o adresie miejsca w pamięci i typie zmiennej tam przechowywanej
- Wskaźnik **`void`** nie posiada wiedzy o typie obiektu na jaki wskazuje
- ```
  void *w1;
  int *w2;
  float *w3;
  w1 = w2;
  w1 = w3;
  w2 = w3; // BŁĄD
  w2 = (int *) w3; // OK
  ```
- Do wskaźnika **`void`** nie można przypisać wskaźnika dowolnego typu, ale z atrybutem **`const`**
  ```

Zastosowanie wskaźników do tablic

- W funkcjach tablice możemy odbierać
  - Jak tablice (`[]`), co jest bardziej czytelne
  - Jako adres (poprzez wskaźnik)
- Zalety i wady
  - W pierwszym przypadku powstaje łatwiejszy do zrozumienia kod
  - W drugim przypadku funkcja działa szybciej, bo operacje na wskaźnikach
- Przykład cpp_3.12

Arytmetyka wskaźników

- Do wskaźników można dodawać i odejmować liczby całkowite – przesuwanie się po tablicy
- Wskaźniki można odejmować od siebie, ale muszą one pokazywać na tą samą tablice. Rezultatem jest odległość między tymi elementami wyrażona liczbą całkowitą
- Wskaźniki można porównywać ze sobą (`==` `!=` `<` `>` `<=` `>=`)
- Przykład cpp_3.13

Wskaźniki do funkcji

- Wskaźniki mogą również pokazywać na funkcje
  - Nazwa funkcji jest jednocześnie jej adresem
  - `void (*f)();`
  - `float (*f)();`
  - `int (*f)(int, int);`
- Przykład cpp_3.14

Rezerwacja obszarów pamięci

- Operacje rezerwacji pamięci wykonuje się za pomocą operatorów **`new`** i **`delete`**
  - **`new`** tworzy nowy obiekt
  - **`delete`** niszczy stworzony wcześniej obiekt
- Cechy
  - Stworzony obiekt nie posiada nazwy, możemy odwoływać się do niego tylko przez wskaźnik
  - Obiekt istnieje od momentu stworzenia do mementu zniszczenie – my decydujemy o tym
  - Obiektów nie obowiązują zwykłe zasady ważności – jeśli mamy wskaźnik pokazujący na dany obiekt to mamy również do niego dostęp
  - Obiekty stworzone za pomocą **`new`** nie są inicjalizowane, musimy sami o to zadbać

new i delete

- Operatory **new** i **delete** zastępują stare funkcje **malloc()** i **free()** z języka C
- Operatory **new** i **delete** są częścią składową języka C++
- ```
  int *a;
  a = new int;
  ...
  delete a;
  ```
- ```
  float *f;
  f = new float[100];
  ...
  delete [] f; //stosowane do tablic!!!
  ```
- Operatorem **delete** kasujemy tylko obiekty stworzone za pomocą **new** i wykonać możemy to tylko raz!!!
- Należy pamiętać, żeby nie „zgubić" wskaźnika do stworzonego i istniejącego obiektu
- Przykład cpp_3.15